Minimizing Strides in Loops with Affine Array References

Philippe Clauss, Vincent Loechner, Benoît Meister
ICPS/LSIIT, Université Louis Pasteur, Strasbourg, France
{clauss,loechner,meister}@icps.u-strasbg.fr

April 25, 2001

Abstract

A significant source of enhancing application performance and of reducing power consumption in embedded processor applications is to improve the usage of the memory hierarchy. In this work, a temporal and spatial locality optimization framework of nested loops is proposed, driven by parameterized cost functions. It is based on the minimization of strides occuring while accessing array elements from an affine reference function. A classification of such strides into two categories induces two joint optimization techniques: data sequence minimization and inter data sequence localization. Unlike many existing methods, our approach pays special attention to TLB (Translation Lookaside Buffer) effectiveness since a TLB miss can take up to three times more cycles than a cache miss. It only considers active data, *i.e.* array elements that are actually accessed by a loop, in order to prevent useless memory loads and take advantage of storage compression and temporal locality. Moreover, the same data transformation is not necessarily applied to a whole array.
Keywords: data locality, memory hierarchy, cache performance, TLB effectiveness, spatial and temporal locality, loop nests, parameterized polyhedron, Ehrhart polynomial.

1 Introduction

Efficient use of memory ressources is a significant source of enhancing application performance and of reducing power consumption in the case of embedded processor applications. Nowadays computers include several levels of memory hierarchy in which the lower levels are large but slow (disks, memory, ...) while the higher levels are fast but small (caches, registers, ...). Hence, programs should be designed for the highest percentage of accesses to be made to the higher levels of memory. To accomplish this, two basic principles have to be considered, both of which are related to the way physical cache and memory are implemented: spatial locality and temporal locality. Temporal locality is achieved if an accessed memory location is accessed again before it has been replaced, since cache lines and pages are kept in memory until they must be replaced with new accessed data. Since a cache miss or page fault for a single data element will bring an entire cache line into cache or page into main memory, if several closely located memory locations are accessed before the cache line or page are replaced, then spatial locality is achieved. Taking advantage of spatial and temporal locality translates to minimizing cache misses, TLB (translation lookaside buffer) misses and page faults.

This classical point of view can be generalized into the framework of *access strides minimization*. Processing arrays with stride-zero corresponds to temporal locality, while stride-one accesses take ideally advantage of spatial locality. However, these two optimizations are not unique. Our measures show that in a more general way, minimizing *all* strides for all data accesses improves performance of a given program. This is particularly significant when a large stride for the accessed data yields to TLB or cache overrun, while the data accessed with reduced stride fits the TLB reach or cache size.

Many works have provided advances in loop and data transformation theory. By using affine representation of loops, several loop transformations have been incorporated into a single framework using a matrix representation of these transformations [18]. These techniques consist either in unimodular [2] or non-unimodular [10] iteration space transformations as well as tiling [18, 17]. Although there has been less attention paid to data transformations, some works investigate the effect of data transformations combined with loop transformations. However, the data transformations they consider are restricted to be unimodular [8, 1, 10].

More recently, Kandemir *et al.* [9] and O'Boyle and Knijnenburg [12] have proposed an unifying framework for loop and more general data transformations. In [12], the authors propose an extension to nonsingular data transformations. Unfortunately, all these approaches do not use any symbolic analysis in order to evaluate the array sizes, the number of reused data, the number of iterations and does not derive symbolic transformations. Moreover, the derived data transformations only consider the innermost loop level and only one unique data layout is associated to each array. Such determined data layouts favor one axis of the index space over the others and neighbors in the unfavored directions become distant in memory [3, 15]. Hence, they can yield large strides when control returns to the enclosing loop, and also yield many useless cache loads of inactive elements. In the same way, the derived loop transformations are dedicated to temporal locality optimization in the innermost loop for one or several references. Then, no more attention is paid to the other references although strides minimization for other data sequences could have been done. Additionally in these recent developments, when a given data layout is determined from one or several references in a first optimized loop, this layout is propagated to all the other references to the same array in the program. Then some loop transformations are needed to take advantage of this layout. But such transformations may not be valid due to data dependences or may be incompatible with some other references. We show that in such cases, it is possible to reduce the strides, for example if there is a significant subset of iterations referencing array elements that are not referenced in the first optimized loop.

In this paper, the transformations are expressed and guided depending on several parameters such as the size of the referenced data set or the parameterized loop bounds. Our framework is based on a classification of strides into two categories:

1. strides resulting from the reuse of a data sequence: when temporal reuse occurs from a given affine array reference, data accesses correspond to several successive reuses of data sequences. Some large strides can then occur while repeating access to the sequence, moving from the last data of the sequence to the first one once again.

2. strides resulting from accessing successively data inside a sequence: data composing a reused sequence maybe distant in memory.

Strides of the second category are minimized by computing new data layouts in order to obtain stride-one access as often as possible. These layouts are defined by new one-dimensional access functions to array elements. Such a function translates to a parameterized data layout depending on the loop bounds or on the array sizes. When it has been evaluated as profitable, different access functions associated to disjoint data subsets are computed for a given array. An interesting consequence is that the reused data sequences will then be composed of data that are contiguous in memory. This approach is the unique one considered in most other works. On the other hand our approach only considers active data and can result in more general data layouts defined by multi-variate polynomials.

Since our data layout transformation ensures that data forming a reused sequence will be contiguous in memory, strides of the first category are minimized by transforming the iteration space in order to minimize the size of the reused data sequences. Strides resulting from repeated accesses to such a sequence will then be minimized since the distance between the first and the last data is reduced. Observe that the best result is obtained while sequences are reduced to one single data. Such a case corresponds to the classical and unique temporal localization objective of most other works.

Since the definition of a new data layout is constrained by the data forming the reused sequences and by their access order, our global optimization framework first considers the first category of strides in order to define the final reused data sequences.

Our techniques use the polytope model [7] with an intensive use of its parametric extensions [6] and of their implementations in the polyhedral library *PolyLib* [16]. The global optimization algorithm driven by parameterized loop cost functions is detailed in section 2. Section 3 presents our loop transformation framework dedicated to minimization of strides of the first category and to previously defined data layout

Nest 1	Nest 2

```
do i = 1,N
 do j = 1,N
  do k = 1,N
   do l = 1,N
    A(k,j,i) = B(l,j)+B(i,l)
   enddo
  enddo
 enddo
enddo
```

```
do i = 1,N
 do j = 1,N
  do l = 1,N
   do k = 1,N
    A(k,j,i) = B(l,j)+B(i,l)
   enddo
  enddo
 enddo
enddo
```

Figure 1: Two different ways to scan the loop nest.

stride size	nest 1	nest 2
$O(N^3)$	0	0
$O(N^2)$	$N^3 + N$	$N^2 + N$
$O(N)$	$N^4 + N^2$	$2(N^3 - N^2)$
1	$N^4 + N^2 - N$	$N^4 + N^2 - N$
0	$N^4 - N^3$	$2N^4 - N^3 - N^2$

Table 1: Number of strides of any size for each loop nest.

adequacy. Our data layout transformation technique for strides minimization of the second category is presented in section 4. The consequences resulting from our optimizations on parallelized code are discussed in section 5. Finally, conclusions and future objectives are given in section 6.

2 Cost criteria driven optimizations of a nested loop

When considering one loop nest, the best combination of loop and data transformations depends on previous optimizations made on costlier nests and on the cost of the occurring references [11]. Between several possible solutions, the best ones are characterized by the smallest strides occurring from their references. For example, let us consider two different possible solutions to scan the loop nest presented in figure 1. For each loop nest and each reference, we compute the number of strides of any size occurring while incrementing the loop indices. The results are given in table 1. For example in the first loop nest, strides of size $O(N^2)$ occur $N^3 + N$ times.

Since the objective is to minimize the number of large strides, solution 1 is eliminated since it yields more strides of size $O(N^2)$ than solution 2. This result has been verified with performance measures.

The best solution should be determined directly without enumerating all the possible ones. But such a direct approach can only be built from heuristics leading to a good solution. The largest stride that a reference i may generate is the size of the set of accessed data $Data_i$. In a general way, some strides larger than one are generated by reusing several times the same data sequence. Such a reuse corresponds to a temporal reuse direction associated with the reference. In order to prevent these large strides, the innermost loop should scan data along this direction. Conversely, the more outer is the loop scanning along this direction, the larger are the strides occurring while incrementing its index. Hence, minimizing stride sizes consists in scanning data along this direction with the most inner possible loop, where temporal reuse will occur on the smallest possible data sequence. Notice that the most favourable case corresponds to the classical temporal optimization where the reused data sequence is reduced to one unique data. All these facts constitute the general concept of *data sequence localization*.

From this concept, we classify strides in two categories: strides occuring from several reuse of a data sequence, and strides occuring from the successive use of data forming a data sequence.

Several references in a loop nest have to be simultaneously considered with several temporal reuse directions. If there exist reuse directions that are common to several references, best solutions are characterized by inner loops scanning along them. The largest data sets are more likely to induce large strides from the first category resulting in many cache and TLB misses. Hence, best solutions are such that the different reuse directions are associated with loops ordered from the innermost to the outermost loop, following the descending size of the associated sets of accessed data $\#Data_i$. Our loop transformation algorithm presented in section 3 is based on these observations.

For any given reference i and its associated iteration space D_i, the size of its data set $\#Data_i$ is given by the Ehrhart polynomial of the affine transformation R_i of D_i, where R_i is the reference matrix [5].

In order to minimize the second category of strides, stride-one access is obtained as often as possible by computing new data layouts as presented in section 4. In the above example, this approach would have lead to the second nest which is the best solution.

3 Loop transformations for strides minimization

This section describes how to perform strides minimization given one loop containing some references to optimize. Each reference is associated with an accessed set of data, and all references are ordered in decreasing data set sizes. Notice that when temporal optimization is achieved, each data accessed by a temporally optimized reference will be stored in a register during the execution of the inner optimized loops. If accessed in read/write or read only, the data is stored in a register at the beginning of the temporal usage, and if accessed in write, it is written to memory at the end of the temporal usage.

Temporal locality or strides minimization is achieved by applying a transformation to the original loop. In this work, we consider unimodular transformations, being equivalent to any combination of loop interchange, skewing and reversal (see [18] for references). In order to be valid, the transformation has to respect the dependences of the loop. This is described in subsection 3.1. An algorithm to compute the transformation matrix for minimizing strides is given in subsection 3.2. This subsection also describes the additional constraints that have to be considered in order to take advantage of the spatial organization of an array accessed in another loop.

3.1 Definitions and prerequisites

We assume that the loops are normalized such that their step is one. The iteration space of a loop nest of depth d is a d-dimensional convex polyhedron, defined by parameterized linear inequalities imposed by the loop bounds. It is denoted by D_P with $P = (p_1, ..., p_q)$ the parameter vector. Each point of the iteration space is denoted by a d-vector $I = (I_1, I_2, ..., I_d)$. Each I_k denotes a loop index with I_1 as the outermost and I_d the innermost loop. We will use $I = (I_1, I_2, ..., I_d)$ to denote an iteration as well as a point in the iteration space.

A reference to an array element is represented by the pair (R, o) where R is the access matrix and o is the offset vector. Elements of R and o are rational numbers. We are also able to handle parameterized references. Therefore elements of o can be affine combinations of integer parameters. A reference is an affine function $r(I) = RI + o$. The references can occur in any loop of the nest since imperfectly nested loops are considered. 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.

Let T be a unimodular $d \times d$ matrix and τ a d-vector. A **unimodular transformation** t of the iteration domain is defined as: $t(I) = TI + \tau$. The transformed domain $D'_P = t(D_P)$ corresponds to a new scanning loop, obtained by applying the Fourier-Motzkin algorithm [2] to D'_P. This algorithm computes the lower and upper bounds of each iteration variable I'_k as a function of the parameters and the variables $I'_1...I'_{k-1}$ only. The body of the loop has also to be transformed in order to use vector I': all references to vector I are replaced by $t^{-1}(I')$.

If the loop nest is not perfect then some more work has to be done. First, we have to compute the set of iteration domains corresponding to the original loop nest. All these iteration domains have to be defined in the same geometrical space (same dimension and same variables). This can be done by using a variant of code sinking [18, 11].Then, the transformation is applied to all these iteration domains. Finally, to get the resulting loop nest we apply Quilleré's algorithm [13], which constructs an optimized loop nest scanning several iteration domains simultaneously.

From this point, by **data dependence**, we will mean flow, anti-, and output dependences only. Input dependence does not play a role, since distinct reads of the same memory location can be done in no particular order. We denote by $\preceq$ the lexicographical *lower or equal* operator.

Let $\mathcal{D}$ be the set of *distance vectors* related to data dependences occurring in the original loop nest:

$$\delta \in \mathcal{D} \Leftrightarrow \begin{array}{l} \exists\, I, J \in D_P, \text{ with } I \preceq J, \text{ such that } J = I + \delta, \text{ and there} \\ \text{is a data dependence from iteration } I \text{ to iteration } J. \end{array}$$

Notice that all distance vectors are lexicographically non-negative. The condition for equivalence between the transformed loop nest and the original loop nest is that [2]: $t(\delta) \succ 0$ for each positive δ in $\mathcal{D}$. Null dependence vectors correspond to loop-independent dependences. A distance vector δ in the original loop nest will become a distance vector $t(\delta)$ in the new one: in order for iteration $J' = t(J)$ to be executed after $I' = t(I)$, vector $t(\delta) = t(J - I) = J' - I'$ must be lexicographically positive.

3.2 Minimizing strides

The main objective of our approach is to reduce stride sizes generated from any reference as far as possible. The largest stride that can be generated by a reference is the size of its data set. Hence, our heuristic chooses references associated with the largest data sets to be optimized priorly.

Let us consider an iteration domain D_P of dimension d, referencing n arrays through reference functions r_i, $1 \leq i \leq n$. We first compute the linear spaces $\mathcal{H}_i$ where temporal reuse occurs for each reference i, as follows. The application of transformation r_i to the iteration space results in a polytope containing all the data accessed by reference i, and is called the data set $Data_i$. Each integer point d_0 of the data set, or each data, corresponds to a polytope to be scanned by the temporal innermost loops if possible. This polytope is parameterized by d_0, and it is computed by applying function preimage[1] r_i^{-1} to d_0, intersected with the iteration domain D_P. The linear space $\mathcal{H}_i$ is the linear hull of this parameterized polytope, *i.e.* it is generated by the basis vectors of the polytope affine hull.

Our algorithm consists in building a basis for the scanning loops represented as matrix B: two successive iterations of a given loop of depth d are linked by the d^{th} column vector of matrix B. The leftmost column corresponds to the scanning of the outermost loop, and the rightmost one to the innermost loop. Matrix B must be full rank and unimodular, and transformation matrix T is equal to B^{-1}. The rightmost scanning vector of B is chosen so that it optimizes as many references as possible. The algorithm then selects the next scanning vectors in order to optimize as many non yet optimized references as possible. In case of equality, the references having the largest data set sizes are chosen priorly. After the initialization, an iterative process generates each scanning vector, from right to left in matrix B:

1. Order the n references in decreasing data set sizes.
2. Compute the linear spaces $\mathcal{H}_i$ for each reference, $1 \leq i \leq n$.
3. for $col = d$ downto 1 do
    4. Find the direction $\mathcal{T}$ that optimizes as many references as possible, in the set of references that have been optimized the least. This is done by computing successive intersections of the sets $\mathcal{H}_i$.

       If there are no more references to optimize choose a direction such that it takes advantage of the spatial organization of a data if necessary, and such that B is unimodular.
    5. Put the vector $\mathcal{T}$ in the column col of matrix B. Check the dependences and the unimodularity of B; if this is not satisfied, go back to step 4 and choose another direction.

       The unimodularity of B is checked as follows: at each step of the algorithm, the $d - col$ vectors generate a linear space; for the final matrix B to be unimodular, each integer point of this linear space should possibly be generated as an integer combination of the $d - col$ vectors. In other words, these vectors must generate a dense lattice subspace. This condition is verified if

[1]Preimage is the inverse function of image. It is implemented in the PolyLib.

original loop nest	transformed loop nest

<pre>do i = 1, N
  do j = i, N
    do k = 1, N
      do l = 1, N
        A[k+l,j] = A[k+l,j] + B[i,j-i]
      enddo
    enddo
  enddo
enddo</pre>
	<pre>do u = 1, N
	  do v = 2, 2N
	    do w = 1, u
	      do x = max(1,v-N), min(N, v-1)
	        A[v,u] = A[v,u] + B[w,u-w]
	      enddo
	    enddo
	  enddo
	enddo</pre>

Figure 2: Example: original and transformed loop nests.

and only if the *gcd* of the subdeterminants of order $d - col$ of these column vectors is 1. This property is based on a corollary of the Hermite normal form theorem ([14], corollary 4.1c).

Example 1 Consider the 4-dimensional loop nest "Nest 1" presented on the left of figure 1 in the previous section. We choose to optimize temporal reuse priorly for reference $A[k,j,i]$, since it has the largest data set (N^3). The two references to B have the same data set size (N^2). Let us number the references to $A[k,j,i]$, $B[l,j]$ and $B[i,l]$ from 1 to 3 respectively. There is only one dependence, on variable A, of distance vector $(0,\ 0,\ 0,\ 1)^{\perp}$.

The first step of the algorithm consists in computing the linear spaces of temporal reuse $\mathcal{H}_i$, for $i = 1, 2, 3$. $\mathcal{H}_1$ is supported by vector $(0,\ 0,\ 0,\ 1)^{\perp}$, $\mathcal{H}_2$ is supported by $(1,\ 0,\ 0,\ 0)^{\perp}$ and $(0,\ 0,\ 1,\ 0)^{\perp}$, and $\mathcal{H}_3$ is supported by $(0,\ 1,\ 0,\ 0)^{\perp}$ and $(0,\ 0,\ 1,\ 0)^{\perp}$.

The algorithm then selects four new scanning vectors in this order:

- vector $(0,\ 0,\ 1,\ 0)^{\perp}$ is chosen first, since it optimizes both references R_2 and R_3 (but not R_1).
- vector $(0,\ 0,\ 0,\ 1)^{\perp}$ optimizes then reference R_1 which has not been optimized yet. Notice that reference R_1 cannot be better optimized in a further step, since $\mathcal{H}_1$ is supported by this vector only.
- vector $(0,\ 1,\ 0,\ 0)^{\perp}$ optimizes reference R_3.
- vector $(1,\ 0,\ 0,\ 0)^{\perp}$ optimizes reference R_2.

Finally, we get: $B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = T$. The dependence is satisfied and the resulting loop is "Nest 2", presented on the right of figure 1. Our experimental results show, after spatial optimization in both cases, a huge improvement of performance: for $N = 700$, the number of TLB's is about $4.9 * 10^9$ for the original loop, versus $8.7 * 10^6$ for the transformed loop, on the R12000 processor. The transformed loop is more than 2 times faster than the original one.

Example 2 Consider the original 4-dimensional loop nest presented on figure 2. There is a dependence for array A, of distance vector $v = (0, 0, 1, -1)^{\perp}$. According to section 2, we choose to optimize temporal reuse priorly for the first reference $A[k+l, j]$, since it has the largest data set ($2N^2 - N$ array elements, versus $\frac{1}{2}(N^2 + N)$ for the second one). $\mathcal{H}_1$ is supported by $(1,\ 0,\ 0,\ 0)^{\perp}$ and $(0,\ 0,\ -1,\ 1)^{\perp}$, and $\mathcal{H}_2$ is supported by $(0,\ 0,\ 1,\ 0)^{\perp}$ and $(0,\ 0,\ 0,\ 1)^{\perp}$.

$\mathcal{H}_1 \cap \mathcal{H}_2$ is supported by vector $(0,\ 0,\ -1,\ 1)^{\perp}$. This is the direction of the innermost scanning loop, where both references are optimized.

We choose as second one, the direction priorly optimizing reference R_1 since the first reference has the largest data set: vector $(1,\ 0,\ 0,\ 0)^{\perp}$. The third one optimizes reference R_2, let us take for example vector $(0,\ 0,\ 1,\ 0)^{\perp}$. The last one will scan the data spaces, let us choose for example $(0,\ 1,\ 0,\ 0)^{\perp}$, in order for B to be full row and unimodular.

Finally we have $B = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$, and $T = B^{-1} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$. The dependence is satisfied. Let us call the new loop indices u, v, w, x, with $u = j$, $v = k + l$, $w = i$, and $x = l$. The new loop in u, v, w, x presented on the right of figure 2 optimally scans both references. The two inner loops access the same array element by the first reference. For the second reference the innermost x-loop accesses the same array element, and the v-loop reuses a subset of the data space corresponding geometrically to a line, scanned by the w-loop.

4 Data layout transformations for strides minimization

In this section, we present our data transformation method. Its objective is to derive new data layouts for the considered references, in order to get minimal stride accesses inside a reused data sequence, for each reference and for a maximum number of loop levels. Such a result is obtained by taking into account only the actually referenced data, *i.e.* the *active data*.

4.1 The active data

Let $Y(I_0)$ be any element of an array Y referenced in a loop nest. The set of iterations referencing this array element $Y(I_0)$ is defined by: $P(I_0) = \{I \in D_P | RI + o = I_0\}$.

If $P(I_0)$ is empty, then $Y(I_0)$ is never referenced by the loop nest and does never have to be loaded in the cache. Its number of reuses is given by the number of integer points in $P(I_0)$, *i.e.* the *Ehrhart polynomial* $EP(I_0)$ of $P(I_0)$ [4, 6].

If $P(I_0)$ is reduced to a single iteration, then array Y has no temporal reuse. Stride-one access for any loop level will then be obtained by indexing any datum with the position of the iteration that references it, relatively to the execution order: if the q^{th} iteration references $Y(I_0)$, then $Y(I_0)$ 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 $Y(I_0)$ is temporally reused, two cases can occur:

- at least the innermost loop has been temporally optimized for the considered reference as presented in section 3: in this case, a new access matrix has been computed which is only applied to the indices of the outermost loops scanning the data space. Stride-one access is then obtained by indexing any datum in the same way by considering only those outermost loops, since temporal optimization resulted in stride-zero access for the remaining innermost loops.
- the innermost loop has not been temporally optimized for the considered reference: in this case, we choose to consider the first iteration referencing $Y(I_0)$ to compute the new data layout, *i.e.* the lexicographic minimum of $P(I_0)$. This minimum is computed using our geometrical tools as described in [6]. Then, any datum is indexed in the same way by considering only iterations that reference such lexicographic minima. Since in the case of temporal reuse, the same data sequence is accessed several times, this choice ensures stride-one access each time such a sequence occurs. Moreover, if this data sequence is small enough to be loaded entirely in the cache, then temporal locality will also be achieved.

The set of active data is defined by: $Data_Y = \{I_0 | I_0 = RI + o, I \in D_P\}$. These data are associated with the points resulting from the affine transformation (R, o) of the iteration space. We have presented in [5] a method allowing to determine the exact number of these active data. This set is generally not a convex polyhedron containing a regular lattice of points since some "holes" may occur irregularly: the image by an affine function of a linearly bounded lattice is *not* a linearly bounded lattice.

In our case however, an exact determination of $Data_Y$ is not necessary. The convex hull $Conv(Data_Y)$, is sufficient in order to reduce the answer of our method to the strictly useful informations. Hence data situated outside this set are assuredly not accessed. We use the PolyLib to compute $Conv(Data_Y)$, by projecting the polyhedron $\{(I, I_0, P) | I_0 = RI + o, I \in D_P\}$ along the I axes.

4.2 Mapping array elements to memory

The lexicographic minimum of the set $P(I_0)$ defines the first iteration referencing any array element $Y(I_0)$. If $P(I_0)$ contains one single point, this point defines the unique iteration referencing $Y(I_0)$. Let I_{min} be this point. The coordinates of $I_{min} = (i_{1,min}, i_{2,min}, ..., i_{n,min})$ are affine functions of I_0. The position of this iteration, relatively to the execution order, gives the position index in main memory of the referenced datum $Y(I_0)$, relatively to the base address b of Y. This will ensure stride-one access since the datum referenced just before or just after $Y(I_0)$ will be contiguous to $Y(I_0)$ in memory.

The position of iteration I_{min} is determined by computing the number of iterations referencing an array element for the first time and occurring before I_{min}. The set of iterations occurring before I_{min} (included) is defined by: $P(I_{min}) = \{I \in D_P | I \preceq I_{min}\}$. where $\preceq$ denotes the lexicographic order. In order to compute the number of iterations in $P(I_{min})$, the lexicographic inequality has first to be transformed into linear inequalities. This transformation will result in a decomposition of $P(I_{min})$ as a union of disjoint convex polyhedra in the following way:

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

where

$$\begin{array}{ll} P_1(I_{min}) & = \{I \in D_P | i_1 < i_{1,min}\} \\ P_2(I_{min}) & = \{I \in D_P | i_1 = i_{1,min}, i_2 < i_{2,min}\} \\ ... & \\ P_n(I_{min}) & = \{I \in D_P | i_1 = i_{1,min}, i_2 = i_{2,min}, ..., i_{n-1} = i_{n-1,min}, i_n \leq i_{n,min}\} \end{array}$$

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

Example 3 In order to show the generality of the technique, let us consider the following example that will result in a rather complicated data layout. It consists in the following loop nest:

```
do i = 1,N
  do j = i,2*i-1
    do k = i-j,i+j
      X = X + Y(i+j-1,i+k+2,j+k)
    enddo
  enddo
enddo
```

Let us focus on array Y. The set of iterations referencing any array element $Y(i0, j0, k0)$ is defined by:

$$\begin{array}{lll} P(i0, j0, k0) & = & \{(i, j, k) \in D_N | i + j - 1 = i0, i + k + 2 = j0, j + k = k0\} \\ & = & \{(i, j, k) \in D_N | i = \frac{i0+j0-k0-1}{2}, j = \frac{i0-j0+k0+3}{2}, k = \frac{-i0+j0+k0-3}{2}\}. \end{array}$$

The set $P(i0, j0, k0)$ contains at most one point defining a unique iteration referencing any array element $Y(i0, j0, k0)$. This iteration is defined by $I_{min} = (\frac{i0+j0-k0-1}{2}, \frac{i0-j0+k0+3}{2}, \frac{-i0+j0+k0-3}{2})$. Hence, the number of iterations occurring before I_{min} is determined by computing the Ehrhart polynomial of the following union of polyhedra:

$$\begin{array}{l} P(\frac{i0+j0-k0-1}{2}, \frac{i0-j0+k0+3}{2}, \frac{-i0+j0+k0-3}{2}) \\ = \{(i, j, k) \in P \mid i < \frac{i0+j0-k0-1}{2}, (i0, j0, k0) \in Conv(Data_Y)\} \\ \cup \left\{ (i, j, k) \in P \left| \begin{array}{l} i = \frac{i0+j0-k0-1}{2}, j < \frac{i0-j0+k0+3}{2}, \\ (i0, j0, k0) \in Conv(Data_Y) \end{array} \right. \right\} \\ \cup \left\{ (i, j, k) \in P \left| \begin{array}{l} i = \frac{i0+j0-k0-1}{2}, j = \frac{i0-j0+k0+3}{2}, k \leq \frac{-i0+j0+k0-3}{2}, \\ (i0, j0, k0) \in Conv(Data_Y) \end{array} \right. \right\} \end{array}$$

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

$$\begin{aligned} EP_1(i0,j0,k0) = \quad & -\tfrac{1}{8}k0^3 + \tfrac{3}{8}j0k0^2 + \tfrac{3}{8}i0k0^2 - \tfrac{3}{4}k0^2 - \tfrac{3}{8}j0^2k0 - \tfrac{3}{4}i0j0k0 + \tfrac{3}{2}j0k0 - \tfrac{3}{8}i0^2k0 + \tfrac{3}{2}i0k0 \\ & -\tfrac{11}{8}k0 + \tfrac{1}{8}j0^3 + \tfrac{3}{8}i0j0^2 - \tfrac{3}{4}j0^2 + \tfrac{3}{8}i0^2j0 - \tfrac{3}{2}i0j0 + \tfrac{11}{8}j0 + \tfrac{1}{8}i0^3 - \tfrac{3}{4}i0^2 + \tfrac{11}{8}i0 - \tfrac{3}{4} \end{aligned}$$

On the second, the answer is: $EP_2(i0,j0,k0) = i0k0 - i0j0 + k0 - j0 + 2i0 + 2$,
and on the third one: $EP_3(i0,j0,k0) = \frac{3}{2}k0 - \frac{1}{2}j0 - \frac{1}{2}i0 + \frac{3}{2}$.

Hence, reference $Y(i+j-1, i+k+2, j+k)$ is transformed into the reference $Y(-i(i(\frac{5}{2}-i)+\frac{1}{2}) + j(j+1) + k + 1)$. The measures we made on the R12000 for $N = 100$ give the following results:

Original loop		
# L1 data cache misses	:	1,012,645
# L2 data cache misses	:	121,679
# TLB misses	:	1,042,231
computation time	:	0.32s

Loop with transformed reference		
# L1 data cache misses	:	127,155
# L2 data cache misses	:	31,234
# TLB misses	:	152
computation time	:	0.02s

□

The array reference function that we provide seems to represent expensive calculations for a compiler. But our many experiments have shown that this is not a significant issue at all, since classical compiler optimizations will reduce it significantly.

When several references to a same array occur in a loop nest, different data layouts can be determined for this array. It is also the case when references to a same array occur in different loop nests. This technique is presented in the following subsection.

4.3 Allocating arrays with different data layouts

For the sake of clarity and due to the lack of space, we describe our technique for cases where only two references to a same array occur in one loop nest. A general presentation can be found in [11].

Let us consider two references $Y(R_1 I + o_1)$ and $Y(R_2 I + o_2)$, $(R_1, o_1) \neq (R_2, o_2)$, that are accessed by scanning the same iteration space D_P. We denote by $Data_1$, respectively $Data_2$, the set active data for reference $Y(R_1 I + o_1)$, resp. $Y(R_2 I + o_2)$: $Data_i = \{I_0 | I_0 = R_i I + o_i, I \in D_P\}$, with $i = 1, 2$.

If $Data_1 = Data_2$, only one data layout that optimizes only one of both references can be determined. Otherwise, some data accessed by one reference are not accessed by the other one. In this case, two different data layouts defined by two Ehrhart polynomials EP_1 and EP_2 are determined for $Data_1 \cup Data_2$. These data transformations can be applied in two ways, leading to different solutions:

- EP_1 is applied to all the data accessed by reference $Y(R_1 I + o_1)$ and EP_2 is applied to the data accessed by $Y(R_2 I + o_2)$ that are not accessed by $Y(R_1 I + o_1)$. In this case, data that are accessed by both references are not accessed with stride-one when accessed by the reference $Y(R_2 I + o_2)$.
- EP_2 is applied to all the data accessed by reference $Y(R_2 I + o_2)$ and EP_1 is applied to the data accessed by $Y(R_1 I + o_1)$ that are not accessed by $Y(R_2 I + o_2)$. In this case, data that are accessed by both references are not accessed with stride-one when accessed by the reference $Y(R_1 I + o_1)$.

The best solution is the one where the defavourable case occurs less: we compute for each solution the number of iterations where stride-one access does not occur for one of the references, and then choose the solution having the minimum number of these iterations. This computation is presented hereunder.

Let $r_i(I) = R_i I + o_i$, for $i = 1, 2$. Then we have $Data_i = r_i(D_P)$. The set of iterations for which the data accessed by $Y(R_2 I + o_2)$ are also accessed by $Y(R_1 I + o_1)$ (for some other set of iterations) is defined by: $S_{12} = r_2^{-1}(r_1(D_P)) \cap D_P$. Since r_2 is not necesseraly invertible, r_2^{-1} denotes the *preimage*, *i.e.* the inverse operation of image: given a domain D', $r_2^{-1}(D')$ is the set of points which, when transformed by r_2, gives D'. It is a convex polyhedron, and this operation is implemented in the polyhedral library *PolyLib* [16]. In the same way, $S_{21} = r_1^{-1}(r_2(D_P)) \cap D_P$ is the set of iterations for which the data accessed by $Y(R_1 I + o_1)$ are also accessed by $Y(R_2 I + o_2)$ for some iterations.

Both sets S_{12} and S_{21} are the iterations for which stride-one access does not occur for one of the references. Hence, the best solution is characterized by the smallest of these sets. For each set, we compute the Ehrhart polynomial giving its number of elements. Suppose that $\#S_{12} < \#S_{21}$. The best solution is the one where stride-one access always occurs for the first reference and occurs for the second reference only when it accesses data that are never accessed by the first reference.

In order to compute both different data layouts and the new loop nest, the original loop has to be split into two different loops $L1$ and $L2$: $L1$ scans the set of iterations S_{12} and $L2$ scans the set $D_P - S_{12}$. The data layout defined by EP_1 is computed from the original loop with the first reference, and the data layout defined by EP_2 is computed from $L2$ with the second reference. EP_1 is then applied to both references in $L1$ and to the first reference in $L2$, and EP_2 is applied to the second reference in $L2$. Moreover, in order to allocate consecutively in memory data accessed by EP_1 and data accessed by EP_2, the greatest index of the data accessed by EP_1 is added to EP_2.

Example 4 Consider the following loop nest:

```
do i = 1,N
  do j = 1,N
    do k = 1,N
     Y = Y + A(k,i) + A(j+k,j)
    enddo
  enddo
enddo
```

Let $r_1(i,j) = (k,i)$ and $r_2(i,j) = (j+k,j)$. We have $D_N = \{(i,j) \in \mathbb{Z}^2 | 1 \le i \le N, 1 \le j \le N, 1 \le k \le N\}$. The set of iterations for which the data accessed by $A(j+k,j)$ are also accessed by $A(k,i)$ is defined by: $S_{12} = r_2^{-1}(r_1(D)) \cap D = \{(i,j) \in \mathbb{Z}^2 | 1 \le i \le N, 1 \le j \le N, 1 \le k \le N-j\}$.

In the same way, the set of iterations for which the data accessed by $A(k,i)$ are also accessed by $A(j+k,j)$ is defined by: $S_{21} = r_1^{-1}(r_2(D)) \cap D = \{(i,j) \in \mathbb{Z}^2 | 1 \le i \le N, 1 \le j \le N, i+1 \le k \le N\}$.

Computations of the Ehrhart polynomials of each set give the following answer:

$$\#S_{12} = \#S_{21} = \frac{1}{2}N^3 - \frac{1}{2}N^2 + N$$

Therefore, both solutions are equivalent. Let us choose the one characterized by S_{12}. The resulting loop nest is obtained by splitting the innermost loop indexed by k into two loops where index k ranges from 1 to $N-j$ for the first one and from $N-j+1$ to N for the second one.

The new data layout for reference $A(k,i)$ is computed from the original loop as described in the previous subsections: $EP_1(i0, j0) = N(j0-1) + i0$. The new data layout for reference $A(j+k,j)$ is computed from the second innermost loop of the new loop nest: $EP_2(i0, j0) = i0(i0-1)/2 + j0$. Since the data with the greatest index accessed by the first reference is $A(EP_1(N,N)) = A(N^2)$, N^2 is added to $EP_2(i0, j0)$. The resulting loop nest is:

```
do i = 1,N
  do j = 1,N
    do k = 1,N-j
     Y = Y + A(N*(i-1)+k) + A(N*(j-1)+j+k)
    enddo
    do k=N-j+1,N
     Y = Y + A(N*(i-1)+k) + A(j+k+(j*j-j)/2+N*N)
    enddo
  enddo
enddo
```

It is important to mention that such a loop splitting can always be done without altering the data dependences, since the initial order of the iterations is not affected at all: any loop inside a considered loop nest is split into several successive loops such that the included instructions are executed in the same order as in the original loop, and without modifying any of the enclosing loops.

5 Consequences on parallel optimizations

Although our method is devoted to improve savings in cache and TLB miss rates, it also has an important impact on processor locality: when a processor brings a data page into its local memory, it will reuse it as much as possible due to our temporal and spatial optimizations. This yields significant reductions in page faults and hence in network traffic.

We can also say, as mentioned by Kandemir *et al* in [9], that optimized programs do not need explicit data placement techniques on shared memory NUMA architectures: when a processor uses a data page frequently, the page is either replicated onto that processor's memory or migrated into it. In either cases, most of the remaining accesses will be local.

Unlike other methods, our data layout transformations prevent processors to bring pages containing useless data that could have been accessed by some other processors. This is due to the fact that only active data are considered in our optimizations.

Temporal optimizations presented in section 3 often generate outer loops that carry no reuse and no data dependences. Hence, these outer loops are perfect candidates for parallelization since they do not share any data.

All these facts allow to generate data-parallel codes with significant savings in interprocessor communication.

Our splitting process presented in subsection 4.3 allows to extract more and different kind of parallelism, *i.e.* control parallelism, from a program: consider two successive loops nests $L1$ and $L2$ that share the same array in an initial sequential program, where the first nest reads some array elements and the second nest writes some other array elements. If the second nest accesses a significative set of array elements that are not accessed by the first nest, it is split into two loop nests $L21$ and $L22$ such that $L22$ accesses elements that are not accessed by the other nests. This latter nest $L22$ can then be computed in parallel with $L1$ since it does not share the same array elements at all.

6 Conclusion

We presented a novel approach based on the concept of reused data sequences for cache and TLB effectiveness. Our method encloses joint temporal and spatial localizations and provides even more efficient solutions, by non-linear data layout definitions and unimodular iteration space transformations.

All the geometric and arithmetic tools used by our method are implemented in the polyhedral library *PolyLib*[2]. Some parts of the global optimizing compiler have already been implemented as code to code transformations, but important developments are still necessary.

Some further improvements can be expected by considering parameters characterizing the target architecture: cache size, cache associativity, smart caches, TLB reach, ... In addition to data locality optimization, some other important issues dedicated to efficient memory use can be considered by this way, such as array padding and cache set conflicts. We now investigate these directions.

References

[1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In ACM Press, editor, *Proc. Principles and Practice of Parallel Programming*, 1995.

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers - The Foundations.* Kluwer Academic Publishers, 1993.

[3] S. Chatterjee, V. V. Jain, A. R. Lebeck, and S. Mundhra. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the ACM International Conference on Supercomputing*, Rhodes, Greece, 1999.

[2] The PolyLib is freely available at http://icps.u-strasbg.fr/Polylib

[4] Ph. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *10th ACM Int. Conf. on Supercomputing*, Philadelphia, 1996.

[5] Ph. Clauss. Handling memory cache policy with integer points countings. In Lengauer, Griebl, and Gorlatch, editors, *Euro-Par'97*, pages 285–293, Passau, August 1997. Springer-Verlag, LNCS 1300.

[6] Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19(2):179–194, July 1998. Kluwer Academic Pub.

[7] P. Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS*, chapter Automatic Parallelization in the Polytope Model, pages 79–100. Springer-Verlag, 1996. G.-R. Perrin and A. Darte, editors.

[8] Y.-J Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformations. In *Proc. 4th Int. Workshop on Languages and Compilers for Parallel Computing.* LNCS 589, Springer Verlag, 1992.

[9] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to global locality optimization. *Journal of Parallel and Distributed Computing*, 58:190–235, 1999.

[10] W. Li. *Compiling for NUMA parallel machines.* PhD thesis, Dept. Computer Science, Cornell University, Ithaca, NY, 1993.

[11] V. Loechner, B. Meister, and Ph. Clauss. Precise data locality optimization of nested loops. Technical report, ICPS, http://icps.u-strasbg.fr, 2001.

[12] M.F.P. O'Boyle and P.M.W. Knijnenburg. Nonsingular data transformations: Definition, validity, and applications. *Int. J. of Parallel Programming*, 27(3):131–159, june 1999.

[13] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Int. J. of Parallel Programming*, 28(5):469–498, october 2000.

[14] A. Schrijver. *Theroy of Linear and Integer Programming.* John Wiley and Sons, New York, 1986.

[15] M. R. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 204–213, june 1998.

[16] D.K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, Rennes, France, 1993. Master's thesis, Oregon State University, Corvallis, Oregon, 1993.

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

[18] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.