To appear in Proc. Europar 98, Southampton, UK, September 1998

Enhancing spatial locality via data layout optimizations

M. Kandemir[1] A. Choudhary[2] J. Ramanujam[3] N. Shenoy[2] P. Banerjee[2]

[1] EECS Dept., Syracuse University, Syracuse, NY 13244 (mtk@ece.nwu.edu)
[2] ECE Dept., Northwestern University, Evanston, IL 60208
({choudhar,nagaraj,banerjee}@ece.nwu.edu)
[3] ECE Dept., Louisiana State University, Baton Rouge, LA 70803 (jxr@ee.lsu.edu)

Abstract. This paper aims to improve locality of references by suitably choosing array layouts. We use a new definition of spatial reuse vectors that takes into account memory layout of arrays. This capability creates two opportunities. First, it allows us to develop an array restructuring framework based on a combination of hyperplane theory and reuse vectors. Second, it allows us to observe the effect of different array layout optimizations on spatial reuse vectors. Since the iteration space based locality optimizations also change the spatial reuse vectors, our approach allows us to compare the iteration-space based and data-space based approaches in terms of their effects on spatial reuse vectors. We illustrate the effectiveness of our technique using an example from the BLAS library on the SGI Origin distributed shared-memory machine.

1 Introduction

In most computer systems, exploiting locality of reference is key to high levels of performance. It is well-known that caching of data whether it is private or shared improves memory latency as well as processor utilization. In fact, increasing the cache hit rates is one of the most important factors in reducing the average memory latency. Although cache hit rates in uniprocessors can be improved by optimizing the organization of the cache such as carefully selecting the cache size, line size and associativity, there are still tasks to be done from software. The 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 [10, 9, 13]; these techniques improve data locality *indirectly* as a result of modifying the iteration space traversal order.

In this paper, we focus on an alternative approach to the data locality optimization problem. Unlike traditional compiler techniques, we focus directly on the data space, and attempt to transform data layouts so that better locality is obtained. We present a data restructuring technique which can improve cache performance in uniprocessors as well as multiprocessor systems. Our technique is based on the concept of reuse vectors [9, 13] and uses linear algebra techniques to help a compiler translate potential reuse in a given program into locality. In our technique, we use data transformations expressed by linear full-rank transformation matrices. We also compare data transformations with linear loop transformations that can be expressed by non-singular transformation matrices. We are particularly interested in optimizing spatial locality, that in turn causes better utilization of the long cache lines which represents the current trend in cache architecture.

This paper is organized as follows. In the next section, we outline the background and motivations for our work. Section 3 presents a memory layout representation based on hyperplane theory from linear algebra. In Section 4, we give a method to compute a reuse vector, for a given layout expressed by hyperplanes. We also illustrate how this new definition of reuse vector can be employed to guide data layout transformations. In Section 5, we present our preliminary results on the `syr2k` example from the BLAS library. In Section 6 we review the related work and conclude the paper in Section 7.

2 Background

We represent each iteration of a loop nest of depth n by a loop iteration vector $\mathbf{I} = (i_1, i_2, ..., i_n)$ where i_k is the value of the kth loop from the outermost. Each reference to an m–dimensional array U in such a loop nest is assumed to be an affine function of the iteration represented by $\mathbf{I}$, i.e., $\mathbf{I}$ is mapped onto data element $A_U\mathbf{I} + \mathbf{o_U}$. Here A_U is an $m \times n$ matrix called the *access matrix* and the m-vector $\mathbf{o_U}$ is called the *offset vector* [13, 9]. Consider the following example.

Example 1: **do** *i = li, ui*
  do *j = lj, uj*
    U(j+1,i-1) = V(i-1,j+1) + W(j-1,i+j+1) + X(j) + Y(i)
  enddo
enddo

Here, for array U, $A_U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, and $\mathbf{o_U} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$. Two references to an array are said to belong to a uniformly generated reference (UGR) set if they have the same access matrix [3].

We say that reuse exists during the execution of a loop nest if the same data element or nearby data elements (e.g., the elements stored consecutively in a column for a column-major array) are accessed more than once. Given a loop nest, it is important to know where data reuse exists. A simple way of detecting the loops which carry reuse is to derive a set of vectors called reuse vectors [13, 9]. The objective here is first to use vectors to represent directions in which reuse occurs, and then to transform the loop nest such that these vectors will have some desired properties, i.e., forms.

Consider a reference with an access matrix A_U and offset vector $\mathbf{o_U}$. Two iterations $\mathbf{I}$ and $\mathbf{J}$ reference the same data element if $A_U\mathbf{I} + \mathbf{o_U} = A_U\mathbf{J} + \mathbf{o_U} \Rightarrow A_U(\mathbf{J} - \mathbf{I}) = \mathbf{0} \Rightarrow A_U\mathbf{r_{U_t}} = \mathbf{0} \Rightarrow \mathbf{r_{U_t}} \in Ker\{A_U\}$ where $\mathbf{r_{U_t}} = \mathbf{J} - \mathbf{I}$. In this case we say that reference A_U exhibits temporal reuse; that is, the same data element is used by more than one iteration; $\mathbf{r_{U_t}}$ is termed as temporal reuse vector and $Ker\{A_U\}$ is referred as temporal reuse vector space [13]. As an example, consider the reference $X(j)$ in Example 1. The access matrix of this reference is $(0, 1)$. Therefore, $\mathbf{r_{X_t}} \in Ker\{(0, 1)\}$; that is, $\mathbf{r_{X_t}} \in span\{(1, 0)^T\}$. This means that there is a temporal reuse carried by the i loop. That is, for a fixed j, successive iterations of the i loop access the same element of array X. For concreteness, we say that $(1, 0)^T$ is the temporal reuse vector for that reference. In general, the loop corresponding to the first non-zero element in a reuse vector is said to carry the associated reuse. It should be noted that all references that belong to a single UGR set have the same access matrix; therefore, they have the same temporal reuse vector. The reuse vector for the reference $Y(i)$ on the other hand is $(0, 1)^T$. Intuitively, a reuse vector such as $(0, 1)^T$ means that the successive iterations of the innermost loop will access the same data element. In a sense, this is a desired reuse vector form; because, the reuse is exploited in the innermost loop where it is most useful. On the other hand, a reuse vector such as $(1, 0)^T$ implies that the same data item is accessed in the successive iterations of the outer loop. If the trip count of the innermost loop is very large then it might be the case that between two uses the data item is flushed from cache. In that case we say that reuse has not converted into locality. The main objective of the compiler-based data reuse analysis can be cast as the problem of transforming programs such that as many reuses as possible will be converted into locality. In our terms, this means to transform programs such that the resulting codes will have desired reuse vector forms.

In addition to temporal reuse, spatial reuse is very important in scientific codes. Assuming a column-major memory layout for arrays, a spatial reuse occurs when the same column of an array is accessed more than once. Consider Example 1 again. Assuming column-major memory layouts for all arrays, we note that successive iterations of *j* loop will access elements in the same column of array U. In that case, we say that array U has spatial reuse with respect to *j* loop. Similarly, successive values of *i* loop access elements of the same column of array V; that is, array V has spatial reuse with respect to loop *i*. It should be noted that spatial reuse is defined with respect to a given memory layout, which is column-major in our case. The spatial reuse exhibited by array W however is slightly more complicated to analyze. For this array, elements of a given column c are accessed if and only if $i + j = c - 1$. In mathematical terms, in order for two iteration vectors $\mathbf{I}$ and $\mathbf{J}$ access elements residing in the same column, they should satisfy the condition $A_{U_s}\mathbf{I} = A_{U_s}\mathbf{J}$ where A_{U_s} is the matrix A_U (for an array U) with the first row deleted [13]. From this condition, since A_{U_s} represents a linear mapping, $A_{U_s}(\mathbf{J} - \mathbf{I}) = \mathbf{0} \Rightarrow \mathbf{J} - \mathbf{I} \in Ker\{A_{U_s}\} \Rightarrow \mathbf{r_{U_s}} \in Ker\{A_{U_s}\}$ where $\mathbf{r_{U_s}} = \mathbf{J} - \mathbf{I}$.

In this case, $\mathbf{r_{U_s}}$ is termed as the spatial reuse vector and $Ker\{A_{U_s}\}$ is referred as spatial reuse vector space [13]. When there is no confusion, we use $\mathbf{r_U}$ instead of $\mathbf{r_{U_s}}$ for the sake of clarity. The loop which corresponds to the first non-zero element of the generator vector of $Ker\{A_{U_s}\}$ is said to carry spatial reuse. In Example 1, for array U, the spatial reuse is carried by j loop. In this example, there exists a spatial reuse vector for each reference and these are $\mathbf{r_U} = (0, 1)^T$, $\mathbf{r_V} = (1, 0)^T$, and $\mathbf{r_W} = (1, -1)^T$. Recall that all these reuse vectors are with respect to column-major memory layout. It is possible that a reference may have more than one spatial reuse vector. In that case, these individual reuse vectors constitute a matrix called spatial reuse matrix, which will be denoted in this paper by R_U for an array U or for a reference to an array U when there is no confusion. Individual spatial reuse matrices from individual references to possibly different arrays constitute a combined reuse matrix [9]. The order of the vectors in the combined reuse matrix might be important. Like Li [9], we assume that they are prioritized from left to right according to the frequency of occurrence.

An important attribute of a reuse vector is its height which is defined as the number of dimensions from the first non-zero entry to the last entry. One way of increasing cache locality is to introduce more leading zeros in the reuse vector, called *height reduction* [9]. In this context, the best possible spatial reuse vector is $\mathbf{r} = (0, 0, \cdots, 0, 1)^T$ meaning that the spatial reuse will be carried by the innermost loop in the nest.

We emphasize the fact that temporal reuse can only be manipulated by iteration space (loop) transformations. Data space transformations cannot directly affect the temporal reuse exhibited by a reference. Since, in this paper we are interested only in data layout transformations, we consider only spatial reuse vectors. In addition, we only focus on self-spatial reuses [13].

3 Hyperplanes: an abstraction for array layouts

This section reviews our work [6] on representing memory layouts of multi-dimensional arrays using hyperplanes. In an m-dimensional data space, a *hyperplane* can be defined as a set of tuples $\{(a_1, a_2, ..., a_m) \mid g_1a_1 + g_2a_2 + ... + g_ma_m = c\}$ where g_1, g_2,...,g_m are rational numbers called hyperplane coefficients and c is a rational number called hyperplane constant [12]. For convenience, we use a row vector $g^T = (g_1, g_2, ..., g_m)$ to denote an hyperplane family (with different values of c) whereas $\mathbf{g}$ corresponds to the column vector representation.

In a two-dimensional data space, we can think of a hyperplane family as parallel lines for a fixed coefficient set and different values of c. An important property of the hyperplanes is that two data points (in our case, array elements) $\mathbf{d}_1$ and $\mathbf{d}_2$ belong to the same hyperplane $\mathbf{g}$ if

$$g^T\mathbf{d}_1 = g^T\mathbf{d}_2. \tag{1}$$

For example, a hyperplane vector such as $(0,1)^T$ indicates that two array elements belong to the same hyperplane as long as they have the same value for the column index (i.e. the second dimension); the value for the row index does not matter.

We note that a hyperplane family can be used to partially define memory layout of an array. For example, in two-dimensional array case, a hyperplane family represented by $(0,1)$ indicates that the array in question is divided into columns such that each column corresponds to a hyperplane with a different c value (hyperplane constant). The data elements that have the same c value (that is, the elements which make up a column) are stored in memory consecutively (ordered by their column indices). We assume in a broader sense that two data elements which lie along an hyperplane with a specific c value have spatial locality. Another way of saying this is that two elements $\mathbf{d}_1$ and $\mathbf{d}_2$ have spatial locality if they satisfy equation (1). Notice that this definition of spatial locality is coarse and does not hold in array boundaries; but it is suitable for our purposes.

In higher dimensions, we need more than one hyperplane. We refer the interested reader to [6] for an in-depth discussion of hyperplane based layout representation. In the remainder of this paper, for the sake of simplicity we mainly focus on two-dimensional arrays. Our results easily extend to higher dimensional arrays.

4 Reuse vectors under different layouts

4.1 Determining spatial reuse vectors

The definition of the spatial reuse vector presented earlier is based on the assumption that the memory layout for all arrays is column-major. In this section, we give a definition of spatial reuse vector under a given memory layout. We start with the following theorem. The reader is referred to [7] for the proofs of all the theorems in this paper.

Theorem 1. *Let $\mathbf{g}$ represent a memory layout for an array U, and $\mathbf{r_U}$ the spatial reuse vector associated with a reference represented by the access matrix A_U. Then, the following equality holds between $\mathbf{g}$, $\mathbf{r_U}$ and A_U:*

$$g^T A_U \mathbf{r_U} = 0 \tag{2}$$

If the memory layout of an array is represented by a matrix L (as in three- or higher dimensional cases), then the equation (2) should be satisfied for each row of L. This theorem gives us the relation between memory layouts and spatial reuse vectors and is very important. Notice that for a given g^T, in general, from $g^T A_U \mathbf{r_U} = 0$, the spatial reuse vector $\mathbf{r_U}$ can always be found as

$$\mathbf{r_U} \in Ker\{g^T A_U\} \tag{3}$$

Let us consider the following example assuming row-major memory layout for all arrays:

Example 2: **do** *i = li, ui*
    do *j = lj, uj*
        do *k = lk, uk*
            U(i+k,j+k) = V(i+j,i+k,j+k) + W(k,j,i)
        enddo
    enddo
enddo

The access matrices for this loop nest are $A_U = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$, $A_V = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$, and $A_W = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$. The row-major layout for 2-dimensional arrays is represented by $g^T = (1, 0)$ and by $L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ for 3-dimensional arrays. Using relation (3), we find the following reuse matrices: $R_U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & -1 \end{pmatrix}$, $R_V = \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix}$, and $R_W = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$. Most of the previous research has concentrated on optimizing $\mathbf{r_U}$ given a fixed memory layout. For instance, if an array is stored in column-major order in memory, i.e., $g^T = (0, 1)$, from equation (2) $g^T A_U \mathbf{r_U} = 0 \Rightarrow A_{U_s} \mathbf{r_U} = 0$; this is the definition of spatial reuse vector used previously [9, 13].

4.2 Reproducing the effects of iteration space transformations

In this subsection, we show how to reproduce the effect of a linear loop transformation by using linear data transformations instead. Li [9] shows that an iteration space transformation T transforms a reuse vector $\mathbf{r_U}$ into $\mathbf{r_U}' = T\mathbf{r_U}$. In this paper, we treat temporal locality as a special case of spatial locality. That is, accessing the same element can be considered accessing the same column (for a column-major layout).

Theorem 2. *Given a 'single' loop nest, if a reference is optimized for spatial locality using an iteration space transformation matrix T, the same effect can be obtained by a corresponding data transformation matrix M.*

Consider the following matrix-multiplication example assuming column-major layout:

```
Example 3:  do i = li, ui
              do j = lj, uj
                do k = lk, uk
                  C(i,j) = C(i,j) + A(i,k) * B(k,j)
                enddo
              enddo
            enddo
```

The spatial reuse matrices are $R_C = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$, $R_A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$, and $R_B = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$. By taking into account the frequency of occurrence of each vector, we can write a combined reuse matrix as $R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. Here the leftmost column has the highest priority whereas the rightmost column has the lowest. In this case, Li's algorithm [9] finds the following matrix to optimize the locality $T = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$. The new reuse matrix is $R' = TR = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$. Notice that the reuse vector of highest priority is optimized very well, i.e., its height is reduced to 1. The transformed program is as follows:

do *u = lu, uu*
  do *v = lv, uv*
    do *w = lw, uw*
      *C(w,u) = C(w,u) + A(w,v) * B(v,u)*
    enddo
  enddo
enddo

In the optimized program, the references $C(w,u)$ and $A(w,v)$ have spatial locality in the innermost loop (w–loop) whereas the reference $B(v,u)$ has temporal locality in the innermost loop. Since, in this paper, we treat temporal locality in a loop as a special case of spatial locality, we conclude that in this program all three references have spatial locality in the innermost loop.

Next we show how to obtain the effect of this transformation with data layout transformations. Notice that after the transformation T, the final spatial reuse matrices for individual references are $R_C{}' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$, $R_A{}' = \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{pmatrix}$, and $R_B{}' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$. Now, from each reuse matrix, we choose the most frequently used reuse vector (considering all reuse vectors in all reuse matrices), and use it as a target reuse vector for the associated array. In this example, the target reuse vector happens to be $(0,0,1)^T$ for all references.

For array C: $$g^T A_C \mathbf{r_C}' = 0 \Rightarrow (g_1, g_2) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0 \Rightarrow (g_1, g_2)^T \in Ker\{(0,0)\}$$

For array A: $$g^T A_A \mathbf{r_A}' = 0 \Rightarrow (g_1, g_2) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0 \Rightarrow (g_1, g_2)^T \in Ker\{(0,1)\}$$

For array B: $$g^T A_B \mathbf{r_B}' = 0 \Rightarrow (g_1, g_2) \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0 \Rightarrow (g_1, g_2)^T \in Ker\{(1,0)\}$$

Thus, the array C can have any layout (say row-major), the array A should be row-major, and the array B should be column-major. Notice that if we assign these layouts, then we have spatial locality for all the references with respect to the innermost loop, a result that has also been obtained by the loop transformation T given earlier.

From the previous discussion, we can conclude that given a single loop nest where each array has one UGR set [3], it is always possible to obtain the locality effect of a linear loop transformation by corresponding linear data transformations.

4.3 Optimizing for the best locality

So far, we have shown that under certain conditions, it is possible to reproduce the effect of a loop transformation by data transformations. By studying the effect of a loop transformation on a spatial reuse vector, we can find a corresponding data transformation which can generate the same effect. Now, we go one step further, and prove a stronger result.

Theorem 3. *Given a loop nest where each array has one UGR set, then it is always possible to optimize the array layouts such that each reference will have spatial locality with respect to the innermost loop.*

Consider the following example with a column-major layout for all arrays.

Example 4: **do** *i = li, ui*
  do *j = lj, uj*
    do *k = lk, uk*
      U(i,j,k) = U(i-1,j,k+1) + U(i,j,k-1) + V(j+1,k+1,i-1)
    enddo
  enddo
enddo

Ignoring the temporal reuses, the spatial reuse matrices for individual arrays are $R_U = (1,0,0)^T$ and $R_V = (0,1,0)^T$. Therefore, the combined reuse matrix is $R = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$. Notice that the first column occurs more frequently (three times to be exact). The best iteration space transformation from the locality point of view (using Li's approach [9]) is $T = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ which would generate $R' = TR = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$. Unfortunately, due to the data dependence involving array U, this transformation is not legal. In search for a second best transformation, we have two options:

For the first option, $T_1 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ which gives $R' = T_1 R = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}$. In the transformed nest, the spatial locality for array U is exploited in the second innermost loop (v–loop); and the spatial locality for array V is exploited in the outermost loop (u–loop).

For the second option, $T_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ which gives $R' = T_2 R = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$. In the transformed nest, the spatial locality for array U is exploited in the outermost loop (u–loop); and the spatial locality for array V is exploited in the innermost loop (w–loop). We note that while T_1 improves the spatial locality of U slightly, T_2 improves the spatial locality for V.

It is easy to see that data layout transformations can easily reproduce the effects of T_1 and T_2 (see [7]). However, neither T_1 nor T_2 (nor their data transformation counterparts) is able to optimize the spatial locality for both arrays in the innermost loop. The reason for the failure of the iteration space transformations is the data dependences which prevent the most desired loop permutation.

We now show that how a different data layout transformation can optimize the same nest. We use the best possible spatial reuse vector as the desired (or target) vector for both the arrays. In other words, $\mathbf{r_U}' = \mathbf{r_V}' = (0,0,1)^T$. For array U, $g^T A_U \mathbf{r_U}' = 0 \Rightarrow (g_1, g_2, g_3) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0$; therefore, $(g_1, g_2, g_3)^T \in Ker\{(0,0,1)\}$ meaning that the array U should have a memory layout such that the last dimension should be the fastest changing dimension. The row-major layout is such a memory layout with the layout matrix $L_U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$. For array V, $g^T A_V \mathbf{r_V}' = 0 \Rightarrow (g_1, g_2, g_3) \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0$; thus, $(g_1, g_2, g_3)^T \in Ker\{(0,1,0)\}$ meaning that the memory layout of array V must be such that the second dimension is the fastest

for each array $U \in \mathcal{U}$
  compute the access matrix A_U *representing* U
  $i = n$
  while $(p(i) = 1)$
    $i = i - 1$
  end while
  $\mathbf{r} = \mathbf{e_i}$
  compute $\mathcal{N} = Ker\{(A_U\mathbf{r})^T\}$
  use the elements of the basis set of $\mathcal{N}$ *as rows of the layout matrix for* U
end for each

Fig. 1. Algorithm for optimizing spatial locality.

changing dimension. An example layout matrix which satisfies that is $L_V = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

In this case, for all the references the spatial reuse will be exploited in the innermost loop. This example clearly shows that the data transformations may be effective in some cases where the iteration space transformations fail.

4.4 Optimizing for shared-memory multiprocessors

In a shared-memory multiprocessor case, we should take into account the issues related to parallelism as well. As a rule, to prevent a common form of false sharing, a loop which carries spatial reuse should not be parallelized [9]. False sharing occurs when two processors access the same coherence unit (at least one of them writes) without sharing a data element [4].

We assume that the parallelism decisions are made by a previous pass in the compilation process, and the information for a loop nest is available to our algorithm as a form of vector $\mathbf{p}$ where $p(i)$ (the i^{th} element) is one if the loop i is parallelized otherwise it is zero. Our memory layout determination algorithm is given in Figure 1. In the figure, $\mathcal{U}$ is the set of all arrays referenced in the nest. The symbol n denotes the number of loops in the nest, and $\mathbf{e_i}$ is a vector will all zero entries except for the i^{th} entry which is 1. The algorithm assumes that there is no conflict (in terms of layout requirements) between references to a particular array. If there is a conflict, then a conflict resolution scheme should be applied [6]. For each array, a subset of all possible spatial reuse vectors starting with the best possible vector are tried. The sequence of trials correspond to $\mathbf{e_n}$,...,$\mathbf{e_2}$,$\mathbf{e_1}$. A vector is rejected as target spatial reuse vector if the associated loop is parallelized. Otherwise $\mathbf{e_i}$ with the largest i is selected as target reuse vector. Once a spatial reuse vector is chosen, the remainder of the algorithm involves only computation of a null set and a set of basis vectors for it. Since optimizing compilers for shared-memory multiprocessors are quite successful in parallelizing the outermost loops, the algorithm terminates quickly; and the spatial locality in the innermost loops is exploited without incurring severe false sharing.

5 Experimental Results

We conducted experiments on an SGI Origin, which is a distributed-shared-memory machine that uses 195MHz R10000 processors, and has 32KB L1 data cache and 4MB L2 unified cache. We used the `syr2k` code (in C) from BLAS to show that our data layout transformation framework

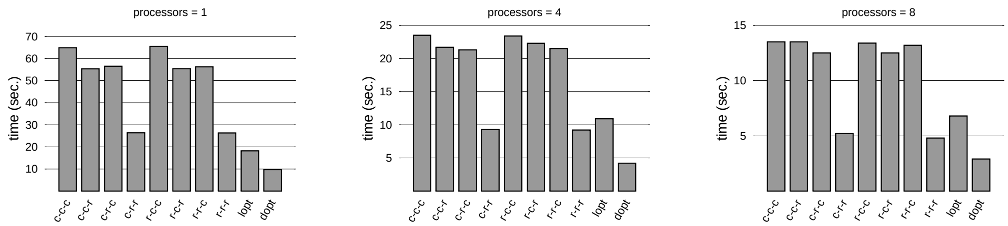

Fig. 2. Execution times (in sec.) of `syr2k` with different layouts on SGI Origin.

can handle complicated access patterns. Assuming a column-major memory layout, all the arrays have poor locality. The version that is optimized by loop transformations (from [9], assuming column-major layouts) optimizes spatial locality for all the arrays, but has two drawbacks: First, the loop bounds and array subscript expressions are very complicated, and second, the temporal locality for array C in the original program has been converted to spatial locality. Our framework, on the other hand, decides suitable memory layouts. Figure 2 shows the performances for eight possible (permutation-based) layouts on an SGI Origin using 1024×1024 double matrices. The legend *x-y-z* means that the memory layouts for C, A and B are *x*, *y* and *z* respectively, where *c* means column-major and *r* means row-major. The layout optimized version –obtained by our framework– (marked `dopt`) and the loop optimized version (marked `lopt`) are also shown as the last two bars. Notice that our framework optimizes the performance substantially, and no dimension re-indexing (permutation-based layout transformation) can obtain this performance.

6 Related Work

The compiler work in optimizing loop nests for locality can be divided into two groups: (1) iteration space based optimizations and (2) data layout optimizations.

In the first group, Wolf and Lam [13] present definitions of different types of reuses and propose an algorithm to maximize locality by evaluating a subset of legal loop transformations. In contrast, Li [9] uses the concept of 'reuse distance'. His algorithm constructs a combined reuse matrix and tries to reduce its 'height' using techniques from linear algebra. McKinley et al. [10] offer a unified optimization technique consisting of loop permutation, loop fusion and loop distribution. Our work on locality is different from those mentioned. First, we focus on data space transformations instead of iteration space transformations. Second, we use a new definition of spatial reuse vector whereas the reuse vectors used in [9] and [13] are oriented for a uniform memory layout.

In the second group, O'Boyle and Knijnenburg [11] focus on restructuring the code given a data transformation matrix and show its usefulness in optimizing spatial locality; in contrast, we concentrate more on the problem of determining suitable layouts by taking into account false sharing as well. Anderson et al. [1] propose a data transformation technique—comprised of permutations and strip-mining—that restructures the data in the shared memory space such that the data for each processor are stored in nearby locations. In contrast, we focus on a larger space of data transformations. Cierniak and Li [2] and Kandemir et al. [5] propose optimization techniques that combine loop and data transformations in a unified framework, but restrict the transformation space. Even in the restricted space, they perform sort of exhaustive search. Jeremiassen and Eggers [4] use data transformations to reduce false sharing. Our framework also considers reducing a common form of false sharing as well. Leung and Zahorjan [8] present an array restructuring framework to optimize locality in uniprocessors. Our work differs from theirs in several points: (1) our technique is based on explicit representation of memory layouts which is central to a unified loop and data transformation framework such as ours; (2) our technique finds optimal memory layouts in a single step rather than first determining a transformation matrix and then refining it for minimizing memory space using Fourier-Motzkin elimination; and (3) we explicitly take false sharing into account for multiprocessors.

7 Conclusions

In this paper, we have presented a definition of reuse vector under a given memory layout. For this, we have represented the memory layout of a multi-dimensional array using hyperplane families. Then we have presented a relation between layout representation and spatial reuse vector. We have shown that this relation can be exploited at least in two ways. For a given layout and reference, we can find the spatial reuse vector *or* for a given desired spatial reuse vector, we can determine the target layout. This second usage allows us to develop a data layout reorganization framework based on the existing compiler technology. Given a loop nest, our framework can optimize the memory layouts of arrays by considering the best possible reuse vectors.

References

1. J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. *Proc. 5th SIGPLAN Symp. Prin. & Prac. Para. Prog.*, Jul. 1995.
2. M. Cierniak, and W. Li. Unifying data and control transformations for distributed shared memory machines. *Proc. SIGPLAN '95 Conf. Prog. Lang. Des.& Impl.*, Jun. 1995.
3. D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations, *J. Para. & Dist. Comp.*, 5:587–616, 1988.
4. T. Jeremiassen, and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *Proc. 5th SIGPLAN Symp. Prin. & Prac. Para. Prog.*, Jul. 1995.
5. M. Kandemir, J. Ramanujam, and A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines. *Proc. 1997 International Conference on Parallel Architectures and Compilation Techniques*, pp. 236–247, Nov. 1997.
6. M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A data layout optimization technique based on hyperplanes. TR CPDC-TR-97-04, Northwestern Univ.
7. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Optimizing spatial locality in loop nests using linear algebra. *Proc. 7th Workshop Compilers for Parallel Computers,* 1998.
8. S-T. Leung, and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report, CSE Dept., University of Washington, TR 95-09-01, Sep. 1995.
9. W. Li. Compiling for NUMA parallel machines. Ph.D. dissertation, Cornell University, 1993.
10. K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems,* 1996.
11. M. 'Boyle, and P. Knijnenburg. Non-singular data transformations: Definition, validity, applications. *Proc. 6th Workshop on Compilers for Parallel Computers*, pp. 287–297, 1996.
12. J. Ramanujam, and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. In *IEEE Trans. Para. & Dist. Sys.,* 2(4):472–482, Oct. 1991.
13. M. Wolf, and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pp. 30–44, June 1991.