# Enhancing Spatial Locality using Data Layout Optimizations

*M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee*

**CPDC-TR-97-07**

December 1997

*Center for Parallel and Distributed Computing*
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

# Enhancing Spatial Locality using Data Layout Optimizations

*M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee*

## 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 two benchmark examples on two distributed shared-memory machines, the Convex Exemplar and the SGI Origin.

## 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 uniprocessor systems can be improved by optimizing the organization of the cache such as carefully selecting the cache size, line (block) size and associativity, there are still tasks to be done from system software point of view. Coherence related misses are a major issue in multiprocessors; therefore, eliminating such misses improves cache performance substantially. Along that direction, fine tuning of the cache coherence protocol and selecting an appropriate line size are very important techniques [13]. Of course, by hardware techniques memory latency can only be reduced so far; and after that, software techniques should be applied. It has already been observed that compiler techniques are useful for optimizing locality in both uniprocessors and multiprocessors, and in reducing the number of coherence related misses [3].

From a software point of view, 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 [17, 15, 20]; 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 [15, 20] and uses linear algebra techniques to help a compiler translate inherent 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. We seek answers to the following questions:

(1) How can memory layouts be represented in a vector/matrix form?

(2) How can we represent spatial reuse vectors under different memory layouts?

(3) What are the effects of memory layout transformations on spatial reuse vectors?

(4) How can the desired spatial reuse vectors be obtained by appropriate memory layout transformations?

The rest of 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 two examples. In Section 6 we revise 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 $\vec{I} = (i_1, i_2, ..., i_n)$ where $i_k$ is the value of the $k$th 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 $\vec{I}$, i.e., $\vec{I}$ is mapped onto data element $A_U \vec{I} + \vec{o_U}$. Here $A_U$ is an $m \times n$ matrix called the *access matrix* and the $m$-vector $\vec{o_U}$ is called the *offset vector* [20, 15]. 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**

As an example, for array $U$, $A_U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, and $\vec{o_U} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

Two references to an array are said to belong to a uniformly generated reference (UGR) set if they have the same access matrix [6].

During the execution of a loop nest, when 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 we say that a reuse occurs. 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 [20, 15]. 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 $\vec{o_U}$. Two iterations $\vec{I}$ and $\vec{J}$ reference the same data element if

$$A_U \vec{I} + \vec{o_U} = A_U \vec{J} + \vec{o_U} \Rightarrow A_U(\vec{J} - \vec{I}) = \vec{0} \Rightarrow A_U \vec{r_{U}}_t = \vec{0} \Rightarrow \vec{r_{U}}_t \in Ker\{A_U\}$$

where $\vec{r_{U}}_t = \vec{J} - \vec{I}$.

In that case we say that reference $A_U$ exhibits temporal reuse; that is, the same data element is used by more than one iteration; $\vec{r_{U}}_t$ is termed as temporal reuse vector and $Ker\{A_U\}$ is referred as temporal reuse vector space [20]. As an example, consider the reference $X(j)$ in Example 1. The access matrix of this reference is $(0,1)$. Therefore, $\vec{r_{X}}_t \in Ker\{(0,1)\}$; that is, $\vec{r_{X}}_t \in span\{(1,0)^T\}$. 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 $\vec{I}$ and $\vec{J}$ access elements residing in the same column, they should satisfy the condition

$$A_{U_s}\vec{I} = A_{U_s}\vec{J}$$

where $A_{U_s}$ is the matrix $A_U$ (for an array $U$) with the first row deleted [20]. From this condition, since $A_{U_s}$ represents a linear mapping,

$$A_{U_s}(\vec{J} - \vec{I}) = \vec{0} \Rightarrow \vec{J} - \vec{I} \in Ker\{A_{U_s}\} \Rightarrow r\vec{_U}_s \in Ker\{A_{U_s}\}$$

where $r\vec{_U}_s = \vec{J} - \vec{I}$.

In that case, $r\vec{_U}_s$ is termed as the spatial reuse vector and $Ker\{A_{U_s}\}$ is referred as spatial reuse vector space [20]. When there is no confusion, we use $r\vec{_U}$ instead of $r\vec{_U}_s$ 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 $r\vec{_U} = (0, 1)^T$, $r\vec{_V} = (1, 0)^T$, and $r\vec{_W} = (1, -1)^T$. 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 [15]. The order of the vectors in the combined reuse matrix might be important. Like Li [15], we assume that they are prioritized from left to right according to the frequency of occurrence. That is, a reuse vector that occurs most frequently is placed into the leftmost position whereas a reuse vector which occurs least frequently is placed into the rightmost position. 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* [15]. In this context, the best possible spatial reuse vector is $\vec{r} = (0, 0, \cdots, 0, 1)^T$ meaning that the spatial reuse will be carried by the innermost loop in the nest. Note that in order to have good cache locality, the loop carrying the spatial reuse should be in the innermost positions.

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, the reuse vectors mentioned so far are associated with self-reuses, i.e., a reuse vector originates from a single reference. There is another class of reuse called group-reuses [20]. There exists a group-temporal reuse if distinct references access the same data element. Similarly, if they refer to the same column of the array, they are said to exhibit group-spatial reuse. Since, the cases where group-spatial reuse introduces an added dimension to the self-spatial reuse vector space are very rare, in this paper, we focus only on self-spatial reuses.

Although the reuse vectors and reuse matrices are appropriate representations to work with, they are based on a uniform layout for all arrays. This characteristic makes them difficult to use in a data restructuring framework where data transformations are applied to arrays in an attempt to obtain desired spatial reuse vectors.

## 3 Hyperplanes: an abstraction for defining memory layouts

This section revises our work [12] 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

Table 1: A few possible layouts and the associated hyperplanes for two-dimensional arrays.

| *row-major* | *column-major* | *diagonal* | *anti-diagonal* |
|---|---|---|---|
| $(1,0)$ | $(0,1)$ | $(1,-1)$ | $(1,1)$ |

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 [7, 19]. 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 $\vec{g}$ corresponds to the column vector representation. Each hyperplane in a hyperplane family has the same $g^T$ but a different $c$. We assume that at least one of the hyperplane coefficients is non-zero.

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) $\vec{d_1}$ and $\vec{d_2}$ belong to the same hyperplane $\vec{g}$ if

$$g^T\vec{d_1} = g^T\vec{d_2} \tag{1}$$

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 $\vec{d_1}$ and $\vec{d_2}$ have spatial locality if they satisfy equation (1). Consequently, all the elements in a column (hyperplane) have spatial locality whereas there is not spatial locality between the elements that lie on different columns. We do not care what the relative orders of the hyperplanes with respect to each other. Notice that this definition of spatial locality is coarse and does not hold in array boundaries; but it is suitable for our purposes. A few possible memory layouts and their associated hyperplanes for two-dimensional case are given in Table 1 for convenience.

In higher dimensions, however, we may need more than one hyperplane. Let us concentrate on a three-dimensional array stored as column-major. For such an array, two data points have spatial locality if they differ only in the first dimension. For instance, data points $(2,3,5)$ and $(7,3,5)$ have spatial locality whereas points $(2,3,5)$ and $(2,4,5)$ do not. We can express this fact by using two hyperplanes: $\vec{g}_1 = (0,0,1)^T$ and $\vec{g}_2 = (0,1,0)^T$. In that case, two data points (array elements) $\vec{d_1}$ and $\vec{d_2}$ have spatial locality if both of the following equations hold

$$g_1^T\vec{d_1} = g_1^T\vec{d_2} \tag{2}$$

$$g_2^T\vec{d_1} = g_2^T\vec{d_2} \tag{3}$$

We can state memory layout in this case by a matrix called layout constraint matrix or layout matrix for short:

$$L = \begin{pmatrix} g_1^T \\ g_2^T \end{pmatrix}$$

Again the order of the rows of $L$ may not be important. An important goal in optimizing spatial locality is to restructure a given program such that two consecutive iteration vectors will access data elements that have spatial locality (in particular consecutive data elements). Notice that with hyperplanes in a two-dimensional data space, we can re-state this fact by saying that (omitting the iteration space boundaries) spatial locality can be improved if two consecutive iterations will access data elements which satisfy equation (1). In three-dimensional case, the data points accessed by two consecutive iterations should satisfy both equation (2) and equation (3). The idea can easily be extended to higher dimensions as well. After expressing reuse vectors under different layouts in the following section, we focus on optimizing loop nests for spatial locality by data layout transformations.

An important class of layouts is the one which corresponds to permutations of array dimensions. For two-dimensional arrays, the only such layouts are row-major and column-major. For three-dimensional arrays, there are six possible layouts and so on. Each such permutation-based memory layout implies an *innermost* or *the fastest changing* dimension whose indices change faster than any other dimension. As an example, for row-major layout, the last dimension is the fastest changing dimension whereas for the column-major layout, it is the first dimension. Our framework, however, considers a much broader class of memory layouts, i.e. the layouts that can be expressed by hyperplanes. This, of course, subsumes the permutation-based memory layouts mentioned.

In the remainder of this paper, for the sake of simplicity we mainly focus on two-dimensional arrays. Our results, however, extend to higher dimensional arrays as well.

# 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. This definition is quite general and uses the memory layout characterization given by hyperplanes. We start with the following lemma.

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

$$g^T A_U \vec{r_U} \quad = \quad 0 \tag{4}$$

**Proof:** See Proof 1 in the Appendix.

If the memory layout of an array is represented by a matrix $L$ (as in three- or higher dimensional cases), then the equation (4) should be satisfied for each row of $L$. This lemma 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 $g^T A_U \vec{r_U} = 0$, the spatial reuse vector $\vec{r_U}$ can always be found as

$$\vec{r_U} \quad \in \quad Ker\{g^T A_U\} \tag{5}$$

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 the references in this loop nest are

$$A_U = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}, A_V = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \text{ and } A_W = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

The row-major layout can be represented by $g^T = (1,0)$ for two-dimensional arrays and by $L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ for three-dimensional arrays. Using relation (5), we derive the following:

$\vec{r_U} \in Ker\{g^T A_U\} \Rightarrow \vec{r_U} \in Ker\{(1,0)\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}\} \Rightarrow \vec{r_U} \in Ker\{(1,0,1)\}$.So, $R_U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & -1 \end{pmatrix}$.

$$\vec{r_V} \in Ker\{{g_1}^T A_V\} \Rightarrow \vec{r_V} \in Ker\{(1,0,0)\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}\} \quad \Rightarrow \quad \vec{r_V} \in Ker\{(1,1,0)\}$$

$$\vec{r_V} \in Ker\{{g_2}^T A_V\} \Rightarrow \vec{r_V} \in Ker\{(0,1,0)\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}\} \quad \Rightarrow \quad \vec{r_V} \in Ker\{(1,0,1)\}$$

Therefore, $R_V = \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix}$.

$$\vec{r_W} \in Ker\{{g_1}^T A_W\} \Rightarrow \vec{r_W} \in Ker\{(1,0,0)\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}\} \quad \Rightarrow \quad \vec{r_W} \in Ker\{(0,0,1)\}$$

$$\vec{r_W} \in Ker\{{g_2}^T A_W\} \Rightarrow \vec{r_W} \in Ker\{(0,1,0)\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}\} \quad \Rightarrow \quad \vec{r_W} \in Ker\{(0,1,0)\}$$

Therefore, $R_W = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$.

Most of the previous research have concentrated on optimizing $\vec{r_U}$ given a fixed memory layout. For instance, if memory layout of a two dimensional array is column-major, which means $g^T = (0,1)$, from equation (4)

$$g^T A_U \vec{r_U} = 0 \Rightarrow A_{U_s} \vec{r_U} = 0$$

which is the definition of spatial reuse vector used previously, for example in [20] and [15].

## 4.2 Reproducing the effects of iteration space transformation

In this subsection, we show how to reproduce the effect of a linear loop transformation by using linear data transformations instead. We particularly focus on loop permutations and illustrate that under some conditions data transformations can easily substitute them. Li [15] shows that an iteration space transformation $T$ transforms a reuse vector $\vec{r_U}$ into $\vec{r_U}' = T\vec{r_U}$. In the following, we present some important results. It should be emphasized that, 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) or the same row (for a row-major layout).

**Lemma 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 $M$.*

**Proof:** See Proof 2 in the Appendix.

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 access matrices for the arrays referenced in this nest are

$$A_C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, A_A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \text{ and } A_B = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Let us first see how the iteration space transformation framework proposed in [15] can optimize this nest. The spatial reuse matrices for the references are

$$R_C = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}, R_A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}, \text{ and } R_B = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$$

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 [15] 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 (it's height is reduced to 1). The transformed program, on the other hand, 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 with respect to the innermost loop.

Next we show how to obtain the effect of this transformation with data layout transformations. In the following discussion we use equation (4).

Notice that after the transformation $T$, the final spatial reuse matrices for individual references are

$$R_C{}' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}, R_A{}' = \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{pmatrix}, \text{ and } R_B{}' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

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. Because, this vector appears in all $R_C{}'$, $R_A{}'$, and $R_B{}'$.

| | | | |
|---|---|---|---|
| For array C: | $g^T A_C \vec{r_C}' = 0 \Rightarrow (g_1, g_2) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0$ | $\Rightarrow$ | $(g_1, g_2)^T \in Ker\{(0,0)\}$ |
| For array A: | $g^T A_A \vec{r_A}' = 0 \Rightarrow (g_1, g_2) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0$ | $\Rightarrow$ | $(g_1, g_2)^T \in Ker\{(0,1)\}$ |
| For array B: | $g^T A_B \vec{r_B}' = 0 \Rightarrow (g_1, g_2) \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0$ | $\Rightarrow$ | $(g_1, g_2)^T \in Ker\{(1,0)\}$ |

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 row-major memory layout for arrays $C$ and $A$, and column-major memory layout for array $B$ (without modifying the original loop order), then we have spatial locality for all the references with respect to the innermost loop, a result that is also obtained by the loop transformation $T$ given earlier.

From the previous discussion, we can conclude the following:

**Lemma 3** *Given a single loop nest where each array has one UGR set [6], then it is always possible to obtain the locality effect of a linear loop transformation by corresponding linear data transformations.*

**Proof:** See Proof 3 in the Appendix. For a stronger result, we refer the reader to [12].

### 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. The spatial reuse vectors constitute a common abstraction that can be manipulated by loop as well as data transformations. By studying the effect of a loop transformation on a 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.

**Lemma 4** *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.*

**Proof:** See Proof 4 in the Appendix.

Consider the following example assuming a default column-major memory 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
```

The access matrices for the arrays referenced in this nest are

$$A_U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } A_V = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

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 [15]) 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 $R' = T_1R = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}$.

The transformed loop nest is as follows:

```
do u = lu, uu
  do v = lv, uv
    do w = lw, uw
      U(v,u,w) = U(v-1,u,w+1) + U(v,u,w-1) + V(u+1,w+1,v-1)
    enddo
  enddo
enddo
```

In this 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 $R' = T_2R = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$.

The transformed loop nest is as follows:

```
do u = lu, uu
  do v = lv, uv
    do w = lw, uw
      U(u,w,v) = U(u-1,w,v+1) + U(u,w,v-1) + V(w+1,v+1,u-1)
    enddo
  enddo
enddo
```

In this 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 $V$. Next we show how to reproduce the effects of $T_1$ and $T_2$ using data layout transformations instead.

For the first option, the resulting reuse vectors for arrays $U$ and $V$ are $\vec{r_U}' = (0,1,0)^T$ and $\vec{r_V}' = (1,0,0)^T$, respectively.

For array $U$,

$$g^T A_U \vec{r_U}' = 0 \Rightarrow (g_1, g_2, g_3) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = 0 \Rightarrow (g_1, g_2, g_3)^T \in Ker\{(0,1,0)\}$$

meaning that the array $U$ should have a memory layout such that the second dimension should be the fastest changing (innermost) dimension. An example layout matrix which satisfies that is $L_U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

For array $V$,

$$g^T A_V \vec{r_V}' = 0 \Rightarrow (g_1, g_2, g_3) \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = 0 \Rightarrow (g_1, g_2, g_3)^T \in Ker\{(0,0,1)\}$$

meaning that the array $V$ should have a memory layout such that the last (third) dimension should be the fastest changing dimension. The row-major layout is such a memory layout with the layout matrix $L_V = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$.

We note that under these layouts array $U$ has spatial reuse in the second loop, whereas the array $V$ has spatial reuse in the outermost loop, a result which has previously been obtained using $T_1$.

For the second option, the resulting reuse vectors for arrays $U$ and $V$ are $\vec{r_U}' = (1,0,0)^T$ and $\vec{r_V}' = (0,0,1)^T$, respectively.

For array $U$,

$$g^T A_U \vec{r_U}' = 0 \Rightarrow (g_1, g_2, g_3) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = 0 \Rightarrow (g_1, g_2, g_3)^T \in Ker\{(1,0,0)\}$$

meaning that the array $U$ should have a memory layout such that the first dimension should be the fastest changing dimension. The column-major layout is such a memory layout with the layout matrix $L_U = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$.

For array $V$,

$$g^T A_V \vec{r_V}' = 0 \Rightarrow (g_1, g_2, g_3) \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0 \Rightarrow (g_1, g_2, g_3)^T \in Ker\{(0,1,0)\}$$

meaning that the array $V$ should have a memory layout such that the second dimension should be the fastest changing dimension. An example layout matrix which satisfies that is $L_V = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

We note that under these layouts array $U$ has spatial reuse in the outermost loop, whereas the array $V$ has the spatial reuse in the innermost loop, a result which has previously been obtained using $T_2$.

As can be seen easily, although result in better code than the original, 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,

$$\vec{r_U}' = \vec{r_V}' = (0,0,1)^T$$

For array $U$,

$$g^T A_U \vec{r_U}' = 0 \Rightarrow (g_1, g_2, g_3) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0 \Rightarrow (g_1, g_2, g_3)^T \in Ker\{(0,0,1)\}$$

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 $V$,

$$g^T A_V \vec{r_V}' = 0 \Rightarrow (g_1, g_2, g_3) \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 0 \Rightarrow (g_1, g_2, g_3)^T \in Ker\{(0,1,0)\}$$

meaning that the array $V$ should have a memory layout such that the second dimension should be the fastest 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.

*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*
  $\vec{r} = \vec{e_i}$
  *compute* $\mathcal{N} = Ker\{(A_U\vec{r})^T\}$
  *use the elements of the basis set of* $\mathcal{N}$ *as rows of the layout matrix for* $U$
*end for each*

Figure 1: Algorithm for optimizing spatial locality.

### 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 [15]. False sharing occurs when two processors access the same coherence unit (at least one of them writes) without sharing a data element. In other words, they share the coherence unit (e.g. cache line, page) but each accesses different elements in it [5, 8].

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 $\vec{p}$ 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 $\vec{e_i}$ 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.[1] 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 $\vec{e_n}$, ..., $\vec{e_2}$, $\vec{e_1}$. A vector is rejected as target spatial reuse vector if the associated loop is parallelized. Otherwise $\vec{e_i}$ 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. The elements of this set represents the layout of the array.

It should be noted that we do not attempt to try all possible spatial reuse vectors. Instead we try only $n$ of them. Since the 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.

It should be emphasized that although we present our technique for a single nest, it can easily be extended for multiple nests by enclosing the nests in question with an outermost loop which iterates only once. This is possible because data transformations are applicable to imperfectly nested loops as well. In that case, form of the parallelization information available to our algorithm should also be changed.

## 5 Experimental Results

We conducted experiments using two C programs on Convex Exemplar and SGI Origin. Convex Exemplar uses 180MHz HP PA-RISC 8000 processors with 1MB data and 1MB instruction cache. The Exemplar is built around the hypernode configuration. Each hypernode can have up to 16 processors, 16GB of memory and a CTI cache of size 256MB. The hand-optimized

[1] We use a strategy based on the loop nesting level followed by loop bounds and so on. Work is under way in determining a good scheme that works well for a variety of benchmark programs.

```
real C(N,N), A(N,N), B(N,N)

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

(a) `matmult`–original code

```
real C(N,2b-1), A(N,2b-1), B(N,2b-1)

do i = 1, N
  do j = i, min(i+2*b-2,N)
    do k = max(i-b+1,j-b+1,1), min(i+b-1,j+b-1,N)
      C(i,j-i+1) = C(i,j-i+1) + A(k,i-k+b)*B(k,j-k+b)
         + B(k,i-k+b)*A(k,j-k+b)
    enddo
  enddo
enddo
```

(b) `syr2k`–original code

```
real C(N,2b-1), A(N,2b-1), B(N,2b-1)

do u = max(2-2b,2-N-b,0),
       min(-2+2b,-2+N+b,-2+2N,-1+N)
  do v = u+2min(-u+1-b,1-b,1-N,-u+1-N),
         u+2min(-u-1+b,-1+b,-u-1+N), 2
    do w = max(1,(u+v)/2+1),
           min(N,(-u+v)/2+N,(u+v)/2+N)
      C((-u-v)/2+w,u+1) = C((-u-v)/2+w,u+1)
               + A(w,(-u-v)/2+b)*B(w,(u-v)/2+b)
               + B(w,(-u-v)/2+b)*A(w,(u-v)/2+b)
    enddo
  enddo
enddo
```

(c) `syr2k`–loop optimized

```
real C(2b-1,N), A(N,2:N+2b-1), B(N,2:N+2b-1)

do i = 1, N
  do j = i, min(i+2*b-2,N)
    do k = max(i-b+1,j-b+1,1), min(i+b-1,j+b-1,N)
      C(j-i+1,i) = C(j-i+1,i)
              + A(k,i+b)*B(k,j+b) + B(k,i+b)*A(k,j+b)
    enddo
  enddo
enddo
```

(d) `syr2k`–data optimized

Figure 2: The codes using in the experiments.

programs are compiled by the native C compiler with -O3 option. The timings are taken using the CXpa performance analyzer. Origin uses 195MHz R10000 processors, 32KB L1 data cache and 4MB L2 unified cache. The programs are compiled by the native C compiler with appropriate data transformations using -O2 option. The timings are taken using the *gettimeofday* routine.

The first program is `matmult`, the classic matrix-multiplication routine shown (not in C syntax) in Figure 2(a). Assuming default column-major array layouts, the original program has temporal locality for array $C$, and spatial locality for array $B$. The locality for array $A$, however, is poor. Li's locality optimization framework [15] re-orders the loops as *j-k-i* starting from the outermost position. Figures 3(a) and 3(b) show the execution times of all possible six loop permutations on Exemplar and Origin respectively with $1024 \times 1024$ double matrices. These figures indicate that, for column-major layout, *j-k-i* is indeed the best order from the locality point of view.

Our framework, on the other hand, assigns row-major layout for arrays $C$ and $A$, and column-major layout for array $B$. Figures 4 and 5 present execution times for all eight (permutation-based) memory layout combinations and loop optimized version (denoted `lopt`) on different processor sizes on Exemplar and Origin respectively. 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. These results show that our framework is successful in determining the optimal memory layout. We note that on both machines the performances of *r-r-c* and *c-r-c* are very close. This is because array $C$ has temporal locality in the innermost loop, so an optimized spatial locality is of secondary importance for it. Another point to note is that the performances of loop optimized version (`lopt`) and data layout optimized version (*r-r-c*) are very close.

The second program is `syr2k` from BLAS [4]. It is a three-deep loop nest which performs banded rank-2k update. We used `syr2k` to show that our data layout transformation framework can handle complicated access patterns as well. The original code is given in Figure 2(b). Assuming a column-major memory layout, all the arrays have poor locality. The version that is optimized by loop transformations (from [15], assuming column-major layouts) is shown in Fig-

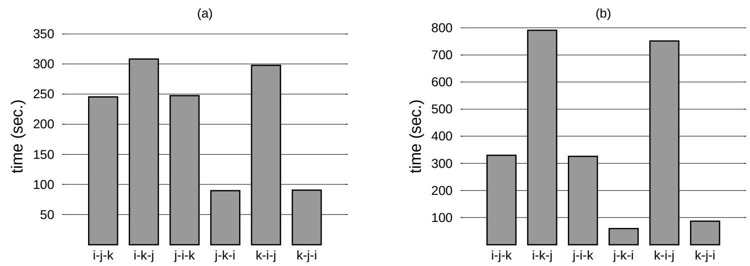


Figure 3: Execution times in seconds for `matmult` with different loop orders on (a) Convex Exemplar and (b) SGI Origin.

ure 2(c).[2] Notice that although the spatial locality is optimized for all the arrays, this version 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 that array $C$ should be row-major, and arrays $A$ and $B$ should be stored along anti-diagonal. We note that although arrays $A$ and $B$ are referenced twice each, that does not cause any conflict in the layout requirements. The layout transformed version is given in Figure 2(d).

Figures 6 and 7 show the performances for eight possible (permutation-based) layouts for Exemplar and Origin respectively for $1024 \times 1024$ double matrices. The layout optimized version –obtained by our framework– (marked `dopt`) and the version given in Figure 2(c) (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.

These two examples also represent different cases. In the `matmult` code, loop and data transformations are equally successful whereas in the `syr2k` code, the data transformations outperform the loop transformation.

## 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 [20] present definitions of different types of reuses and propose an algorithm to optimize locality. Their algorithm evaluates a subset of legal loop transformations and transforms the loop nest such that the locality is maximized. They focus on tiling of the innermost loops. In contrast, Wei Li [15] uses the concept of 'reuse distance'. His algorithm constructs a combined reuse matrix and tries to reduce its 'height' using techniques from linear algebra. The algorithm can represent the reuse vectors precisely and the transformations are operated directly on reuse vectors. McKinley *et al.* [17, 2] offer a unified optimization technique consisting of loop permutation, loop fusion and loop distribution. An important characteristic of their method is that it can optimize for parallelism without sacrificing locality.

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 [15] and [20] are oriented for a uniform memory layout. Unlike [20] and [17], our approach does not evaluate a large space of feasible transformations. Instead, our method detects data layouts by directly working on spatial reuse vectors.

In the second group, O'Boyle and Knijnenburg [18] focus on restructuring the code given a data transformation matrix. They also show their method can be used for optimizing spatial locality. In comparison, 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 for distributed-shared-memory architectures. Their approach restructures the data in the shared memory space such that the data for each processor are stored in nearby locations. They restrict the data transformations to permutation and strip-mining. In contrast, we focus on a larger space of data transformations; that is, the transformations that can be expressed by

[2] It should be emphasized that in porting the program to C the necessary layout transformations were applied.

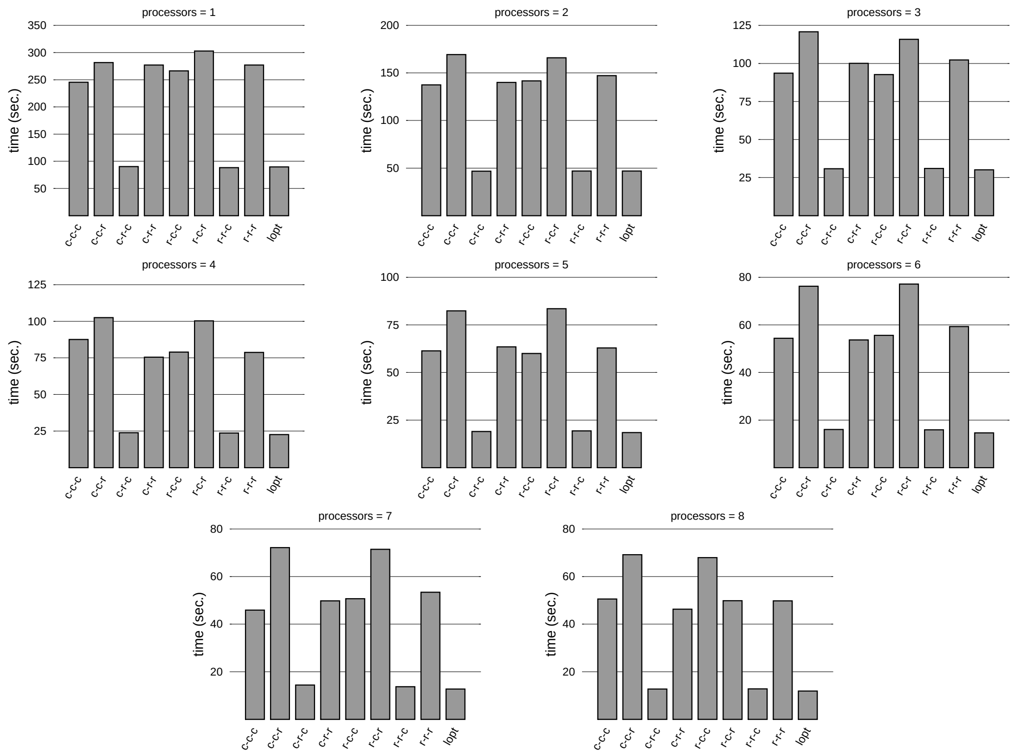


Figure 4: Execution times (in sec.) of `matmult` with different layouts on Convex Exemplar.

hyperplane families. Cierniak and Li [3] and Kandemir *et al.* [10, 11] propose data layout optimization techniques. Their techniques combine loop and data transformations in a unified framework, but restrict the transformation space. In [3], both data and iteration space transformations are restricted whereas in [10] and [11] only data transformations are restricted. Even in the restricted space, they perform sort of exhaustive search. Ju and Dietz [9] present a systematic approach that integrates data layout optimizations and loop transformations to reduce cache coherence overhead. Since their work focuses on reducing coherence misses, it is not directly comparable to ours. Jeremiassen and Eggers [8, 5] also use data transformations. They focus on techniques to reduce false sharing while we focus on optimizing spatial locality so the transformations that they use are different from ours. Our framework however also considers reducing a common form of false sharing as well. Leung and Zahorjan [14] present an array restructuring framework to optimize locality in uniprocessors. Our work differs from theirs in several points: First, our technique is based on explicit representation of memory layouts. This is important because we plan to embed an iteration space transformation approach in our framework. Under such a unified framework, we will be able to transform a loop nest assuming (say) a diagonal memory layout. Second, 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. Finally, we explicitly take false sharing into account in a multiprocessor environment.

## 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

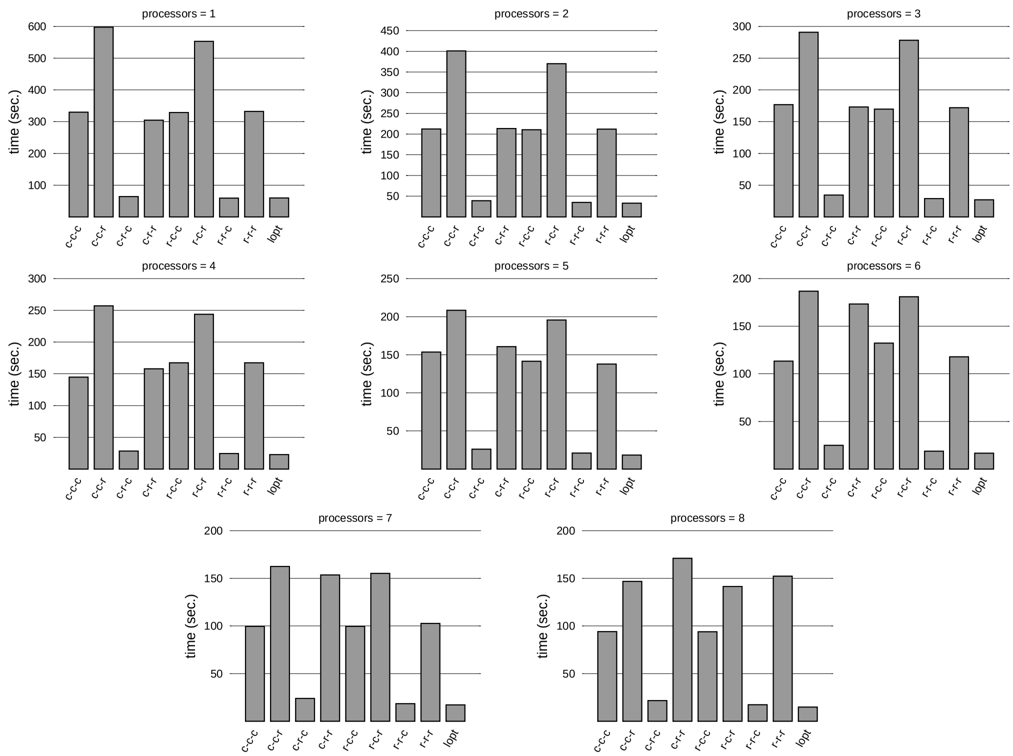


Figure 5: Execution times (in sec.) of `matmult` with different layouts on SGI Origin.

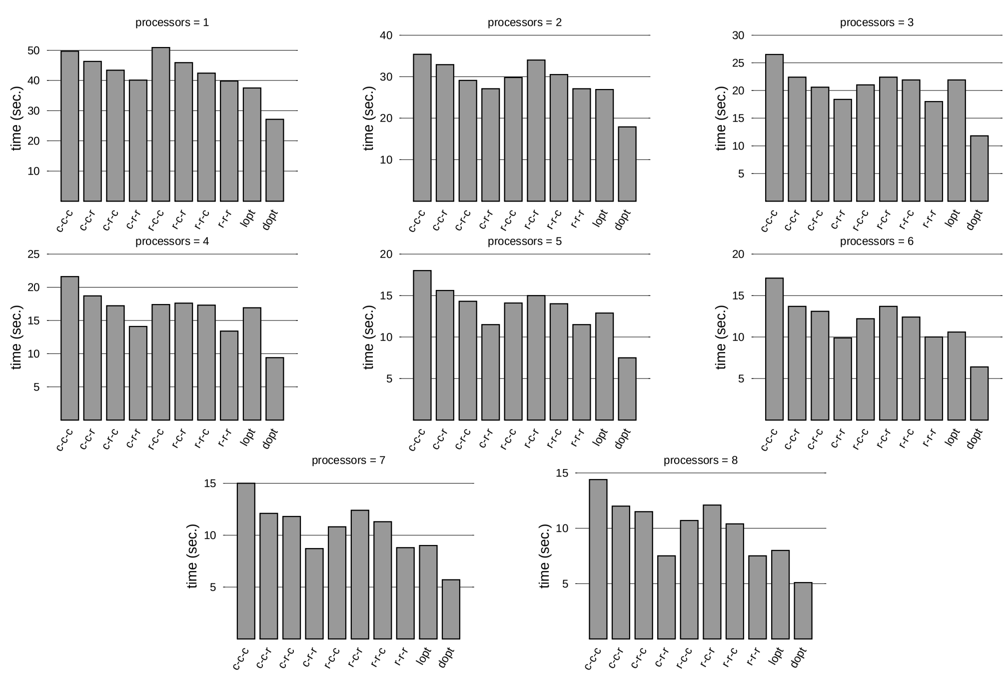


Figure 6: Execution times (in sec.) of `syr2k` with different layouts on Convex Exemplar.

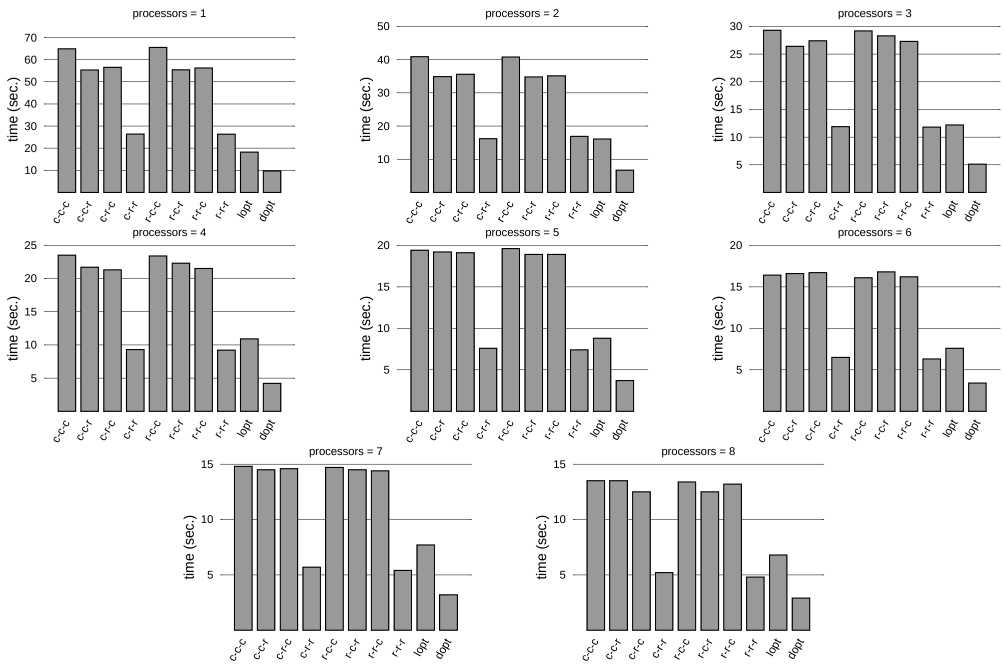


Figure 7: Execution times (in sec.) of `syr2k` with different layouts on SGI Origin.

framework can optimize the memory layouts of arrays by considering the best possible reuse vectors. We plan to extend this work in two steps. First, we will describe an iteration space transformation framework which is customized to optimize spatial locality. Second, we will develop a unified framework which can apply data and iteration space transformations in concert. The salient features of this framework will be its efficiency in deriving non-singular iteration space transformations as well as its capability to take into account any type of layout that can be represented by hyperplane families.

# References

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

[2] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, Oct 1994.

[3] M. Cierniak, and W. Li. Unifying data and control transformations for distributed shared memory machines. *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995.

[4] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. In *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[5] S. J. Eggers, and T. E. Jeremiassen. Eliminating false sharing. In *Proc. the 1991 International Conference on Parallel Processing*, Aug 1991.

[6] 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.

[7] C.-H. Huang, and P. Sadayappan. Communication-free partitioning of nested loops. In *Journal of Parallel and Distributed Computing*, 19:90–102 (1993).

[8] T. E. Jeremiassen, and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[9] Y.-J. Ju, and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Proc. Fourth International Workshop in Languages and Compilers for Parallel Computing*, pages 344–358, Ithaca, NY, 1992.

[10] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proc. 11th ACM International Conference on Supercomputing*, pages 269–276, Vienna, Austria, July 1997.

[11] M. Kandemir, J. Ramanujam, and A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines. In *Proc. 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT 97)*, pages 236–247, San Francisco, CA, November 1997.

[12] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A data layout optimization technique based on hyperplanes. Technical Report, CPDC-TR-97-04, Northwestern University, December 1997.

[13] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: implementation and performance. In *Proc. of the 19th International Symposium on Computer Architecture*, pages 92–103, Gold Coast, Australia, May 1992.

[14] S-T. Leung, and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report, Dept.of Computer Science and Engineering, TR 95-09-01, Sept 1995.

[15] W. Li. Compiling for NUMA parallel machines. Ph.D. dissertation, Cornell University, Ithaca, NY, 1993.

[16] W. Li. Compiler cache optimizations for banded matrix problems. In *Proc. 9th ACM International Conference on Supercomputing*, pages 21–30, Barcelona, Spain, July 1995.

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

[18] M. F. P. O'Boyle, and P. M. W. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers*, pages 287–297, Aachen, Germany, 1996.

[19] J. Ramanujam, and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. In *IEEE Transactions on Parallel and Distributed Systems*, 2(4), pages 472–482, Oct 1991.

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

[21] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, CA, 1996.

[22] H. Zima, and B. Chapman. *Supercompilers for parallel and vector computers*, ACM Press, NY, 1991.

## Appendix: Proofs of results

**Proof 1** Let $\vec{I}$ and $\vec{J}$ be two iteration vectors. The data elements accessed by these vectors using $A_U$ are $A_U\vec{I}$ and $A_U\vec{J}$ respectively. By equation (1), these two elements have spatial locality if

$$g^T A_U \vec{I} = g^T A_U \vec{J} \Rightarrow g^T A_U(\vec{J} - \vec{I}) = 0 \Rightarrow g^T A_U \vec{r_U} = 0$$

where $\vec{r_U} = \vec{J} - \vec{I}$. □

**Proof 2** (sketch) Suppose that $\vec{r_U}$ is the original spatial reuse vector and after the transformation $\vec{r_U}' = Tr_U$. Then $\vec{r_U}'$ is the new reuse vector. Suppose that $\vec{g}$ represents the memory layout that gives us the reuse vector $\vec{r_U}'$. Recall that for matrices $X$ and $Y$, $(XY)^T = Y^T X^T$; then,

$$g^T A_U \vec{r_U}' = 0 \Rightarrow (A_U \vec{r_U}')^T \vec{g} = 0 \Rightarrow \vec{g} \in Ker\{A_U \vec{r_U}'\}.$$

This means that any vector from $Ker\{A_U \vec{r_U}'\}$ is the desired hyperplane vector which represents the memory layout to obtain $\vec{r_U}'$ as the spatial reuse vector.

In that case, assuming $\vec{g_d}$ represents the default memory layout for the array in question, any non-singular data transformation matrix $M$ which satisfies $\vec{g_d}M = \vec{g}$ can be used to obtain the spatial reuse vector $\vec{r_U}'$ (see [12] on how to obtain $M$ matrix and code generation) □

**Proof 3** Obvious, as it is always possible to optimize for any target reuse vector. □

**Proof 4** (sketch) We know that under given conditions for each array $U$, the equality $g^T A_U \vec{r_U} = 0$ holds, where $\vec{r_U}$ is the spatial reuse vector. On the other hand, the best possible spatial reuse vector for array $U$ is $\vec{r_U} = (0, 0, \cdots, 0, 1)^T$. Then,

$$g^T A_U \vec{r_U} = g^T A_U (0, , 0, \cdots, 0, 1)^T = g^T \vec{a_m} = 0 \Rightarrow a_m \vec{g} = 0 \Rightarrow \vec{g} \in Ker\{a_m\}$$

where $\vec{a_m}$ is the last column of $A_U$ and $a_m$ is the row-vector form of $\vec{a_m}$. We can always choose a $\vec{g}$ from $Ker\{a_m\}$. Therefore, it is always possible to find a memory layout corresponding to the best spatial reuse vector which causes the spatial locality exploited in the innermost loop in the nest. □