Final version of paper to appear in IEEE Trans. Parallel and Dist. Systems, Feb 1999

A Linear Algebra Framework for Automatic Determination of Optimal Data Layouts*

M. Kandemir[†‡] A. Choudhary[§‡] N. Shenoy[§‡] P. Banerjee[§¶] J. Ramanujam[||]

Abstract

This paper presents a data layout optimization technique for sequential and parallel programs based on the theory of hyperplanes from linear algebra. Given a program, our framework automatically determines suitable memory layouts that can be expressed by hyperplanes for each array that is referenced. We discuss the cases where data transformations are preferable to loop transformations and show that under certain conditions a loop nest can be optimized for perfect spatial locality by using data transformations. We argue that data transformations can also optimize spatial locality for some arrays without distorting temporal/spatial locality exhibited by others. We divide the problem of optimizing data layout into two independent subproblems: (1) determining optimal static data layouts, and (2) determining data transformation matrices to implement the optimal layouts. By postponing the determination of the transformation matrix to the last stage, our method can be adapted to compilers with different default layouts. We then present an algorithm that considers optimizing parallelism and spatial locality simultaneously. Our results on eight programs on two distributed-shared-memory multiprocessors, the Convex Exemplar SPP-2000 and the SGI Origin 2000, show that the layout optimizations are effective in optimizing spatial locality and parallelism.

Keywords: data reuse, locality optimizations, spatial locality, memory performance, parallelism, array restructuring.

Corresponding Author:

Professor Prith Banerjee
Department of Electrical and Computer Engineering
Northwestern University
L 497, Technological Institute
2145 Sheridan Road
Evanston, IL 60208-3118
Phone: (847) 491-4118, Fax: (847) 467-4144
Email: banerjee@ece.nwu.edu

*A preliminary version of this paper appears in the Proceedings of the *1998 ACM International Conference on Supercomputing (ICS'98).*

†Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244. e-mail: mtk@ece.nwu.edu

‡Supported in part by NSF Young Investigator Award CCR-9357840, NSF grant CCR-9509143, and Air Force Materials Command under contract F30602-97-C-0026.

§Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. e-mail: {choudhar,nagaraj,banerjee}@ece.nwu.edu

¶Supported in part by NSF under grant CCR-9526325 and in part by DARPA under contract DABT-63-97-C-0035.

||Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. e-mail: jxr@ee.lsu.edu.
Supported in part by NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422.

1 Introduction

On most modern parallel machines, the accesses to a nearby memory location are much faster than accesses to a farther location. This encourages programmers and compiler writers to modify the access patterns of a program so that a majority of accesses are made to nearby memory locations. Previous research on optimizing compilers has generally been on iteration space transformations and scheduling techniques to improve locality. Among the techniques used are unimodular [5] and non-unimodular [26, 34] iteration space transformations, tiling [43], loop fusion [30] and affinity scheduling [1]. These techniques focus on improving data locality *indirectly* as a result of modifying the iteration space traversal order.

In this paper, we take a more direct approach to the data locality optimization problem. Unlike traditional compiler techniques, we focus directly on the data space, and attempt to change the data layouts so that better locality is obtained. There are several observations that motivate our approach.

- Some programs are not amenable to loop transformations. Data dependences [44] in a loop nest may not allow a loop transformation to improve locality. On the other hand, data space transformations are not affected by and do not place any restrictions on the data dependences; thus, in principle, they have wider applicability.
- For some programs, even though an iteration space transformation is legal, there may be a data space transformation which results in better locality.
- While loop transformations affect all the arrays accessed in a given loop nest (sometimes adversely), data transformations do not.
- Imperfectly nested loops and explicitly parallelized loops are in general more difficult to optimize using loop transformations whereas in many cases data transformations can be successfully applied to the arrays referenced in such loops.
- These two transformation techniques can be combined in a unified way. In fact, it has already been reported [8, 21, 22] that some programs are best improved by a unified transformation approach.

Based on these observations, we present a framework that uses data transformations to optimize locality. It should be emphasized that both the problem of finding optimal data layouts [19, 20] and that of finding optimal data distributions [23] are NP-complete. Mace [28] has shown that the problem of finding optimal data storage patterns for parallel processing is also NP-complete. For parallel machines, our framework attempts to find optimal memory layouts under static array distributions. The layouts that we find may not be optimal under dynamic array distributions across processors. Our framework is fairly general in the sense that it works on a large search space and considers various memory layouts that can be expressed by hyperplanes. Specifically, in this paper we make the following contributions:

- We show how the hyperplane theory can be used for optimizing cache locality.
- A method that determines the optimal layouts for all arrays referenced in a given a loop nest is presented.
- It is shown that for a single loop nest, under specific conditions, data layout transformations can optimize the nest for perfect locality.

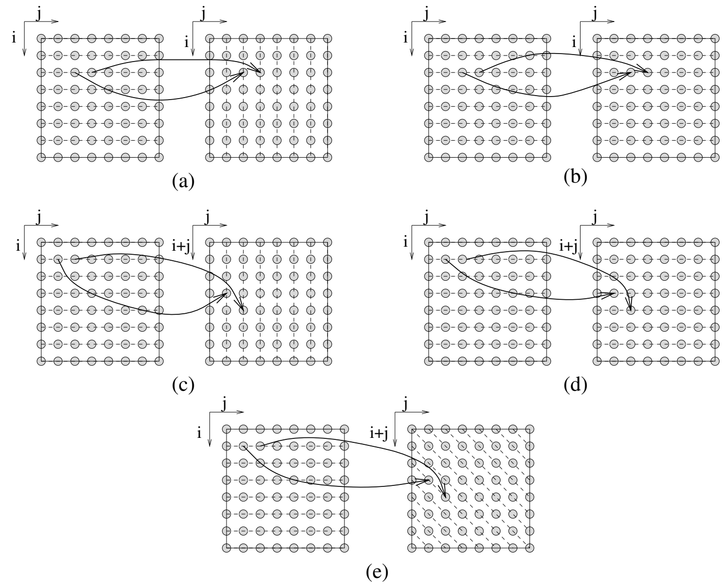

Figure 1: Different access patterns on two-dimensional iteration and data spaces. Two consecutive iterations access (a) two different columns; (b) the same row; (c) the elements in a diagonal (different columns); (d) the elements in a diagonal (different rows); (e) two consecutive elements in the same diagonal. [In each figure, the rectangular shape on the left denotes iteration space whereas that on the right denotes data space].

- We discuss how our method can optimize multiple loop nests simultaneously and how it can be used to improve locality taking into account parallelism.
- We present experimental results on two distributed-shared-memory machines, namely the Convex Exemplar and the SGI Origin 2000 to validate our theoretical results.

In this paper, when we talk about loop transformations we mean linear transformations of the iteration space that can be expressed by square non-singular transformation matrices [44]. Similarly, the data transformations we consider can be expressed as linear non-singular square transformation matrices. It should be noted that data transformations may be useful in other areas besides optimizing for spatial locality. For example, the shift transformation can prove useful in dealing with alignment of data on page or cache line boundaries [31]. In this paper, we do not consider the shift type of layout transformations.

The remainder of this paper is organized as follows. In Section 2 we outline the notation used and review concepts such as reuse and locality. Section 3 presents an overview of hyperplane theory, and Section 4 shows how to use that theory to optimize memory layouts of arrays referenced in loop nests. In Section 5, we present a layout optimization algorithm specifically designed for shared memory multiprocessors. In Section 6, we explain how to obtain a suitable data transformation matrix to generate optimized code. In Section 7, we present experimental results on deriving optimal data layouts for the Convex Exemplar and the SGI Origin. Section 8 discusses the interaction between data distribution and memory layouts on distributed shared memory (DSM) systems. In Section 9, we discuss the related work and conclude the paper with a summary in Section 10.

2 Technical Preliminaries

We view the iteration space of an n-deep loop nest as an n-dimensional polyhedron where each point is denoted by an $n \times 1$ column vector $\bar{I} = (i_1, i_2, ..., i_n)^T$; here, each i_k denotes a loop index with i_1 as the outermost loop and i_n the innermost. In this paper we will use $(i_1, i_2, ..., i_n)$ to denote a loop nest as well as a point in the iteration space. We show the lower and upper limits for a loop `i` as `li` and `ui` respectively. We assume that the loops are normalized such that the step size is one; also, we assume that all the loop bounds and subscript expressions are affine functions of enclosing loop indices. Many regular scientific programs exhibit such structure. Thus, the polyhedron corresponding to the iteration space is bounded by linear inequalities imposed by the loop bounds. Similarly, every array declared in the program defines a polyhedron, each point of which represents an array element; the bounds for this polyhedron are constants and are derived from the array declaration statements. Using these representations of iteration and data spaces, a reference to an array in such a loop nest can be represented by the pair $(A, \bar{o})$ where A is the access (or reference) matrix and $\bar{o}$ is the offset vector [43, 26]. Essentially, such a reference is an affine mapping $\bar{f}(\bar{I}) = A\bar{I} + \bar{o}$, where $\bar{I}$ is the iteration vector. For example, for the reference $X(i-2j, j+3)$ occurring in a two-deep loop nest (i, j), we have

$$A = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix} \text{ and } \bar{o} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}.$$

In general, for a reference to an m-dimensional array inside an n-dimensional loop nest, the access matrix is of size $m \times n$ and the offset vector is m-element vector. An important class of references is the class of *uniformly generated references (UGR)*, first defined by Gannon et al. [10].

Definition 1 *Two references $(A_1, \bar{o}_1)$ and $(A_2, \bar{o}_2)$ to the same array are said to be uniformly generated if $A_1 = A_2$.*

An important characteristic of uniformly generated references from the spatial locality point of view is that the memory layout determination process need to be carried out only once for each set of such references.

In order to obtain high levels of performance from programs running on a sequential or parallel machine that contains some sort of cache memory hierarchy, cache locality should be exploited. That is, a datum brought into the cache should be reused as much as possible before it is replaced. The reuse of the same data while it is still in the cache is termed as *temporal locality*, whereas the use of the nearby data in a cache line is called *spatial locality* [43]. We stress that a program may reuse data, but if that data has been replaced between reuses, we say that it does not exhibit *locality*. Consider the example shown below.

```
do i = li, ui
  do j = lj, uj
    U(j) = V(i) + W(j,i) + X(i,j) + Y(i+j,j)
  enddo
enddo
```

Assuming a column-major memory layout as the default (as for example in Fortran), in this loop nest array U has temporal reuse in the `i` loop and spatial reuse in the `j` loop. Array V has temporal reuse in the `j` loop and spatial reuse in the `i` loop. Array W has only spatial reuse in the `j` loop. Similarly arrays X and Y have only spatial reuses in the `i` loop. Assuming that the trip count (number of iterations) for both the loops is large, only the reuses associated with the `j` loop will exhibit locality during execution. Therefore, the exploitable reuses for this nest are the temporal reuse for V, and the spatial reuses for U and W. A large portion of the previous compiler research has focused on locality. Some of the previous research along that direction will be discussed in Section 9. For this example, interchanging [44] two loops will improve the spatial locality for array X, but destroy the spatial locality for array W. Alternatively, if array X is stored in memory as row-major, without loop interchange the spatial locality will be exploited for both W and X. Figure 1(a) shows the original access pattern assuming a column-major layout for array X. The rectangular shape on the left denotes iteration space whereas that on the right denotes data space. The data accessed by two consecutive iterations fall into different columns, which causes the spatial locality to be poor for this reference. Instead, as shown in Figure 1(b), converting the memory layout of this array into row-major causes two consecutive iterations to access the same row. The situation for array Y, however, is more complicated, as neither a column-major (Figure 1(c)) nor a row-major (Figure 1(d)) layout storage improves locality. Instead, this array should be stored diagonally in memory so that two consecutive iterations access elements on the same diagonal as shown in Figure 1(e).

In this paper, we are interested in deriving data transformations for different arrays accessed in a loop nest such that the innermost loop exhibits maximum spatial locality.

Definition 2 *Two iteration points $\bar{I} = (i_1, i_2, ..., i_n)$ and $\bar{J} = (j_1, j_2, ..., j_n)$ are said to have "proximity in time" if for all k $(1 \leq k \leq n-1)$, $i_k = j_k$.*

Under this definition, for a two-dimensional iteration space given by (i, j) and bounded by $1 \leq i \leq 20$ and $1 \leq j \leq 20$, iterations $(2, 3)$ and $(2, 10)$ have proximity in time whereas iterations $(2, 20)$ and $(3, 5)$ do not. It should be noted that this definition of proximity in time is coarse-grained and does not hold in the boundaries of the iteration space. But as will be shown later, it is very suitable for our purposes. While optimizing the loop nests for spatial locality we will concentrate only on two successive iterations of the innermost loop and omit the bounds of the iteration space.

3 Overview of Hyperplanes

In an m-dimensional space, a *hyperplane* can be defined as a set of tuples

$$\{(a_1, a_2, ..., a_m) \mid g_1 a_1 + g_2 a_2 + ... + g_m a_m = c\}$$

where g_1, g_2,...,g_m are rational numbers called hyperplane coefficients and c is a rational number called hyperplane constant [16, 33, 36]. A hyperplane vector $(g_1, g_2, ..., g_m)$ defines a *hyperplane family* where each member hyperplane has the same hyperplane vector but a different c value. For convenience, we use a row vector $\bar{g}^T = (g_1, g_2, ..., g_m)$ to denote such a hyperplane family whereas $\bar{g}$ corresponds to the column vector representation of the same hyperplane family. When there is no confusion we use g^T instead of $\bar{g}^T$.

We say that two data points (array elements) $\bar{d}_1$ and $\bar{d}_2$ (in a multi-dimensional array) belong to the same data hyperplane $\bar{g}$ if

$$g^T \bar{d}_1 = g^T \bar{d}_2. \quad (1)$$

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

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

For example, in a two-dimensional array space, a hyperplane vector such as $(0, 1)$ 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.

Two data elements may belong to more than one hyperplane as well. For example, in a three-dimensional array space, two data elements may belong to a hyperplane $(0, 0, 1)$ as well as to another hyperplane $(0, 1, 0)$.

Definition 3 *Two data points $\bar{d}_1$ and $\bar{d}_2$ are said to have "proximity in space" or "spatial locality" for a given data hyperplane g^T if equation (1) holds for them.*

As an example, let us focus on hyperplane family defined by $g^T = (0, 1)$. Such a hyperplane family defines column hyperplanes on the data space (see Figure 1(a)). An array element (a, b) belongs to a column (hyperplane) with constant c if and only if $b = c$. In that case, we can say, for instance, that the array elements $(3, 4)$ and $(5, 4)$ have spatial locality (because they are on the same hyperplane) whereas the elements $(3, 4)$ and $(3, 5)$ do not. In other words, as long as the two elements have the same value for the column index they have spatial locality. As with the previous definition, this spatial locality notion is coarse-grained and does not hold at the array boundaries. Notice that, if we do not care about the relative order of hyperplanes, we can use the hyperplane $(0, 1)$ to denote column-major memory layout in a two-dimensional data space. A few possible memory layouts and their associated hyperplane vectors for two-dimensional case are given in Table 1.

In a two-dimensional space, if we ignore the relative order of hyperplanes, a single hyperplane family is sufficient to define a memory layout partially. In three- or higher-dimensional cases, we may have to take into account more than one hyperplane families. As explained later, two data elements in a three-dimensional array stored as column-major have spatial locality if they have spatial locality with respect to $(0, 0, 1)$ *and* $(0, 1, 0)$; that is, if they have the same indices except for the first dimension. The idea can be generalized to higher dimensions as well.

With our definitions of "proximity in time" (for iteration space) and "proximity in space" (for data space) we are now ready to give our locality definition for a loop nest.

Definition 4 *Ignoring the loop and array bounds, a loop nest has "spatial locality" for a given reference R if two iteration points that have proximity in time access data points (both using R) that have proximity in space.*

Definition 5 *Ignoring the loop and array bounds, a loop nest has "perfect spatial locality" if it has "spatial locality" for each reference R that it encloses.*

It should be noted that our definition of spatial locality is broader than the usual meaning of the word as used by previous researchers. We also note that our spatial locality definition is only with respect to the innermost loop in the nest. Finally, we only consider self-spatial reuses (i.e., reuses that originate from a single reference). If a spatial reuse originates from distinct references, we call it group-spatial reuse [43]. 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.

4 Optimizing Spatial Locality by Using Data Layout Transformations

4.1 Problem Definition

Traditionally, arrays are stored in either of two forms: row-major (as in C) and column-major (as in Fortran). Each such representation implies a 'fastest changing' or 'innermost' dimension; that is, a dimension whose index changes faster than any other dimension in a given sequence of consecutive elements. As an example, for a multi-dimensional array stored in row-major, the last dimension is the fastest changing dimension. This fact can be seen by simply observing a sequence of elements stored in memory such as ... (2,3), (2,4), (2,5) ... for a two-dimensional row-major array. Similarly, for a multi-dimensional column-major array, the first dimension is the fastest changing dimension. This storage scheme can easily be generalized by selecting a dimension and making it the fastest changing dimension, then selecting a second fastest changing dimension, and so on. In this way, an m-dimensional array can be stored in memory in $m!$ different ways, each corresponding to a pre-determined ordering of dimensions. Although such a scheme is quite general, it does not handle all possible memory layouts such as diagonal (skewed) layouts. An example that requires a diagonal layout is the reference $Y(i+j,j)$ in the loop nest given above. Assuming that the default layout is column-major in this example, the layout of array Y should be skewed along the diagonal as shown in Figure 1(e) for the best locality to be obtained; that is, the elements on a diagonal should be contiguously stored in memory. The relative order of diagonals with respect to each other is of secondary importance provided that the array sizes are large enough in each dimension. Of course, for a reader, concluding that the skewed layout is optimal for this case is easy; but the general question of determining the required data layouts automatically for the arrays referenced in a given loop nest is an open research problem. This paper addresses this problem and offers an automatic strategy based on the current parallelizing compiler technology and hyperplane theory discussed earlier for determining the suitable memory layouts and data transformations to obtain them. We divide the problem of optimizing locality into two separate sub-problems:

(1) Determination of optimal memory layouts that are defined by hyperplanes; and

(2) Data space transformations to obtain (or implement) optimal layouts.

Each sub-problem can be solved independently. Previously, [8], [21], [22] and [25] offered algorithms to handle the first sub-problem whereas [31] and [25] offered methods to handle the second problem. In fact, the second problem arises because there is no way of specifying the array layouts in conventional languages like Fortran and C.

We note that since a one-dimensional array can only be stored in one way (using linear layout mappings), the data space transformations are only meaningful for two or higher dimensional arrays. It should also be stressed, however, that in some cases, data layout transformations can be applied to one-dimensional arrays as well provided that the arrays can be de-linearized first [29].

Our main objective in this paper is to solve the first subproblem mentioned above, namely, finding optimal memory layouts for each array referenced in a single nest. Once the compiler decides a suitable layout for each array, it is a mechanical process to find the corresponding data transformation matrices to implement the chosen layouts in a given compiler. We choose to separate the problem of determining optimal layout from the problem of finding a suitable data transformation to implement it. The reason for this decision is to make our framework easy to adapt to languages with different default layouts as well as to have explicit memory layout representations.
The problem we address in this paper is defined as follows:

> Given a program in which a number of arrays are accessed, what are the suitable memory layouts for each array such that the loop nests in the program will have spatial locality (as defined earlier) with respect to each reference that they enclose? If this is not possible, we want to maximize the number of references for which this is possible.

Chandra et al. [6] indicate that due to some conditions related to storage and sequence assumptions about the arrays and to passing arrays as subroutine arguments, data transformations may not always be legal. We assume no such situation occurs for the example programs given in this paper. Chandra et al. [6] also propose methods on how to cope with storage sequence and parameter passing problems when data transformations are to be applied. Investigating these issues is beyond the scope of this paper.

4.2 Determining Optimal Layouts

In this subsection, we present our data space restructuring framework in detail. In particular, we show how to determine optimal layouts for different arrays referenced in a loop nest. We start by observing that given large array bounds and trip counts for the enclosing loops, the spatial behavior of a reference is largely determined by the innermost loop index. Let us now concentrate on two consecutive iterations $\bar{I}$ and $\bar{I}_{next}$ of a given loop nest of depth n. These two iterations have identical values for each loop index except for the innermost loop, i.e.,

$$\bar{I} = \begin{pmatrix} i_1 \\ \vdots \\ i_{n-1} \\ i_n \end{pmatrix} \text{ and } \bar{I}_{next} = \begin{pmatrix} i_1 \\ \vdots \\ i_{n-1} \\ 1 + i_n \end{pmatrix}$$

In order to exploit the locality for a reference R denoted by the access matrix A_R, two consecutive iterations $\bar{I}$ and $\bar{I}_{next}$ as defined above should access two data elements that have spatial locality in the data space. In particular, we want the accessed elements to be neighbors so that they can reside on the same (or at least neighboring) cache line(s). We can now give the following result.

Lemma 1 *A given loop nest of depth n exhibits spatial locality with respect to a reference R (denoted by an $m \times n$ access matrix A_R) to an m-dimensional array, if, for each vector $\bar{g}$ defining the memory layout,*

$$\bar{g} \quad \in \quad Ker\{a_n\}, \tag{2}$$

where a_n is the row vector form of the last column of A_R.

Proof: Two data elements accessed by $\bar{I}$ and $\bar{I}_{next}$ are $A_R\bar{I}+\bar{o}$ and $A_R\bar{I}_{next}+\bar{o}$ respectively. From equation (1), in order to have spatial locality

$$g^T(A_R\bar{I}+\bar{o}) = g^T(A_R\bar{I}_{next}+\bar{o}) \Longrightarrow g^T(A_R\bar{I}) = g^T(A_R\bar{I}_{next}) \Longrightarrow g^T(A_R(\bar{I}_{next}-\bar{I})) = 0$$

Since $\bar{I}_{next}-\bar{I} = (0,\cdots,0,1)^T$, we have $g^T(A_R\left((0,\cdots,0,1)^T\right)) = 0 \Longrightarrow g^T\bar{a_n} = 0 \Longrightarrow a_n\bar{g} = 0 \Longrightarrow \bar{g} \in Ker\{a_n\}$ □

Whenever we can find such a hyperplane $\bar{g}$, we use one such that $\gcd\{g_1, g_2, ..., g_m\}$ is the smallest. Consider the following loop nest.

```
do i = li, ui
  do j = lj, uj
    U(i,j) = V(j,i) + W(i+j,i) + X(i+j,j) + Y(n-j,i+j)
  enddo
enddo
```

Assuming that the default layout is column-major for all arrays, the only exploitable spatial reuses in this loop nest are due to arrays V and W. Intuitively, for the optimal cache performance from this loop nest, U should be row-major, arrays V and W should be column-major, X should have a diagonal layout, and array Y should have anti-diagonal layout. We now show how to determine these layouts automatically. In this example, the access matrices are

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

Using Lemma (1) given above, $\bar{g_U} \in Ker\{(0,1)\} \Longrightarrow {g_U}^T = (1,0)$; $\bar{g_V} \in Ker\{(1,0)\} \Longrightarrow {g_V}^T = (0,1)$; $\bar{g_W} \in Ker\{(1,0)\} \Longrightarrow {g_W}^T = (0,1)$; $\bar{g_X} \in Ker\{(1,1)\} \Longrightarrow {g_X}^T = (1,-1)$; and $\bar{g_Y} \in Ker\{(-1,1)\} \Longrightarrow {g_Y}^T = (1,1)$. From Table 1 we see that these vectors represent the optimal layouts for this example.

Notice that, our approach as explained so far is superior to those presented in [21] and [8] as neither of those approaches can detect skewed (diagonal) layouts. Diagonal layouts are useful for banded-matrix operations in general. Although some of the banded matrix applications can be optimized using loop transformations [26], they become so at the expense of complex loop bounds and complex array subscript expressions. The `syr2k` code from the BLAS library [9] is a typical example of that. These complex bounds and subscript expressions incur lots of run-time overhead in turn. In two-dimensional data spaces, a single hyperplane family (denoted by $\bar{g}$) is sufficient to describe the memory layout of an array (provided that the relative orders of hyperplanes are not important). In higher dimensions however, we may need to use more than one hyperplane family to describe a memory layout. In the following subsection, we concentrate on this issue and take a different look at data layouts.

4.3 Layout Relations

Let us for a moment focus on the layout of array X found in the previous example. Such a layout implies that the elements $X(i,j)$ and $X(i+1,j+1)$ should be stored consecutively in memory in order to obtain the best spatial locality. This fact can also be observed if we consider the spatial positions of two data elements (d_1, d_2) and (d'_1, d'_2)

under such a layout. In order for these two data points to have spatial locality, the following equality should hold

$$(1, -1) \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} = (1, -1) \begin{pmatrix} d'_1 \\ d'_2 \end{pmatrix}.$$

This means $d_1 - d_2 = d'_1 - d'_2$. We refer to this equality as *layout relation*. In fact, each layout relation corresponds to a hyperplane and imposes a constraint between two data points; if those two data points satisfy that constraint then they are said to have spatial locality with respect to the associated hyperplane. In this example, since $(3, 4)$ and $(3, 5)$ do not satisfy the constraint given above, they do not have spatial locality. On the other hand, the elements $(3, 4)$ and $(4, 5)$ satisfy the constraint, therefore have spatial locality.

We next consider the following three-dimensional loop nest which accesses three and two dimensional arrays.

```
do i = li, ui
  do j = lj, uj
    do k = lk, uk
      U(j,k,i-2) = V(k,i-1,j+k) + W(k,i+k)
      X(i,j,k) = Y(i+j,i+k,j+k) - 1
    enddo
  enddo
enddo
```

The access matrices for the arrays referenced in this nest are

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

Let us first concentrate on array V. Consider two data points (d_1, d_2, d_3) and (d'_1, d'_2, d'_3). Using Lemma (1),

$$\bar{g_V} \in Ker\{(1, 0, 1)\} \Longrightarrow {g_V}^T = (0, 1, 0) \text{ or } {g_V}^T = (1, 0, -1).$$

Consequently, we have two layout relations: $d_2 = d'_2$ and $d_1 - d_3 = d'_1 - d'_3$. These two layout relations, *together*, define the memory layout for this array. We can view each layout relation (constraint) as defining a *locality group* and the intersection of the constraints defines a smaller locality group. To have a good spatial locality, the elements in this smaller locality group should be stored in consecutive memory locations. This can be seen from Table 2 where the elements of the array V accessed for some representative iterations are shown. We note that each group in the table satisfies both $d_2 = d'_2$ and $d_1 - d_3 = d'_1 - d'_3$. For example, from the first group $V(1, 1, 2)$ and $V(2, 1, 3)$ should be stored in memory together as they satisfy both of the constraints. As long as the arrays are large enough, it is sufficient to store the elements that satisfy both of the constraints as a locality group, provided that these two constraints do not conflict with each other. The relative order of these locality groups (if desired) are determined by considering a larger locality group. In this example, $d_2 = d'_2$ denotes a larger locality group which, for example, contains $V(1, 1, 2)$ and $V(1, 1, 3)$. Similarly, the other constraint, $d_1 - d_3 = d'_1 - d'_3$ also determines a locality group which, for example, includes $V(1, 1, 2)$ and $V(1, 2, 2)$. The choice between those locality groups is made by considering the second column of the access matrix which corresponds to the second innermost loop in the nest. We are now ready to give the following lemma.

Table 2: Three locality groups for reference $V(k, i-1, j+k)$ appearing in (i, j, k). Any two data elements in a column have spatial locality with respect to each other.

Locality group I		Locality group II		Locality group III	
(i,j,k)	$V(k,i-1,j+k)$	(i,j,k)	$V(k,i-1,j+k)$	(i,j,k)	$V(k,i-1,j+k)$
$(2,1,1)$	$V(1,1,2)$	$(2,2,1)$	$V(1,1,3)$	$(3,1,1)$	$V(1,2,2)$
$(2,1,2)$	$V(2,1,3)$	$(2,2,2)$	$V(2,1,4)$	$(3,1,2)$	$V(2,2,3)$
$(2,1,3)$	$V(3,1,4)$	$(2,2,3)$	$V(3,1,5)$	$(3,1,3)$	$V(3,2,4)$
$(2,1,4)$	$V(4,1,5)$	$(2,2,4)$	$V(4,1,6)$	$(3,1,4)$	$V(4,2,5)$
...	...	...	...	...	...

Lemma 2 *A loop nest of depth n exhibits spatial locality in the k^{th} innermost loop with respect to a reference R (denoted by an $m \times n$ access matrix A_R) to an m-dimensional array, if, for each vector $\bar{g}$ defining the memory layout,*

$$\bar{g} \quad \in \quad Ker\{a_{n-k+1}\}, \tag{3}$$

where a_{n-k+1} is the row vector form of the column $n-k+1$ of A_R.

Proof: The proof is very similar to that of Lemma (1); therefore is omitted. □

In our example, two data points (d_1, d_2, d_3) and (d'_1, d'_2, d'_3) have spatial locality in the second innermost (middle) loop if

$$\bar{g_V} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} = \bar{g_V} \begin{pmatrix} d'_1 \\ d'_2 \\ d'_3 \end{pmatrix}.$$

Here $\bar{g_V} \in Ker\{(0,0,1)\} \Longrightarrow {g_V}^T = (0,1,0)$ or ${g_V}^T = (1,0,0)$.

Therefore, we have two locality relations: $d_2 = d'_2$ and $d_1 = d'_1$. Since, $d_2 = d'_2$ does exist as a constraint for the innermost loop as well, we consider it as the *dominant* relation when we order the locality groups in memory with respect to each other. That is, for our example, the larger locality group will include, say, $V(1,1,2)$ and $V(1,1,3)$. But, for example, $V(1,1,2)$ and $V(1,2,2)$ will go to different locality groups. We can now state the resulting layout for this array as

$$\begin{aligned} d_2 &= d'_2 \\ d_1 - d_3 &= d'_1 - d'_3. \end{aligned}$$

The two relations here together give the constraints that two data items should satisfy in order to have spatial locality. The elements that have spatial locality constitute a locality group. We can think of such a locality group as intersection of the elements in two hyperplanes in the data space. The order of relations on the other hand define relative ordering among these locality groups in memory. The set of constraints above, which define the memory layout of array V, can also be expressed as a matrix

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix},$$

where each row defines a hyperplane and corresponds to a layout constraint. We refer to this matrix as *layout constraint matrix* and denote it by L_V for an array V. In our example, the layout constraint matrices are as follows:

$$L_U = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, L_V = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix}, L_W = \begin{pmatrix} 1 & -1 \end{pmatrix}, L_X = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \text{ and } L_Y = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{pmatrix}.$$

Notice that L_X corresponds to row-major layout. Now consider array Y since it demonstrates a different behavior. At the innermost level,

$$\bar{g_Y} \in Ker\{(0,1,1)\} \Longrightarrow {g_Y}^T = (1,0,0) \text{ or } {g_Y}^T = (0,1,-1).$$

Therefore the two elements (d_1, d_2, d_3) and (d'_1, d'_2, d'_3) in a locality group should satisfy the constraints

$$\begin{aligned} d_1 &= d'_1 \\ d_2 - d_3 &= d'_2 - d'_3. \end{aligned}$$

For the relative ordering of the constraints we look at the second innermost loop

$$\bar{g_Y} \in Ker\{(1,0,1)\} \Longrightarrow {g_Y}^T = (0,1,0) \text{ or } {g_Y}^T = (1,0,-1),$$

which means

$$\begin{aligned} d_2 &= d'_2 \\ d_1 - d_3 &= d'_1 - d'_3. \end{aligned}$$

Since the two group of relations have no common relation, compiler chooses an arbitrary order between the relations $d_1 = d'_1$ and $d_2 - d_3 = d'_2 - d'_3$. In general, the approach works as follows:

- First, compute the relations corresponding to innermost loops and divide the array elements into locality groups such that two data elements are put in the same locality group if and only if they satisfy all locality constraints.
- Then, to determine the relative order among locality groups, look at the second innermost loop. If the locality groups can be ordered by doing so, use the derived order; otherwise choose an order arbitrarily.
- If the relations cannot be ordered by considering the second innermost loop, it is possible to continue with the next outer loop, and so on; but our experience shows that in practice this hardly makes a difference.

4.4 Imperfectly Nested Loops

In this subsection we consider imperfectly nested loops which are in general difficult to optimize by loop transformations. Consider the following example.

```
do i = li, ui
  do j = lj, uj
    ...  U(i+j,j)  ...
```

```
        do k = lk, uk
          ...  U(k,j+k) ...
        enddo
      enddo
    enddo
```

In this example, the array U is referenced in two different nesting levels. The access matrices are

$$A_{U_1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \text{ and } A_{U_2} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}.$$

Since, for optimizing spatial locality in the innermost loop, we are only interested in the last columns of the access matrices, and in this case both of the last columns are the same, there is no conflict between the two references, and the array can be stored in memory diagonally. Essentially, the observation is that as far as the data transformations are concerned, the imperfectly nested loops case is no different from the perfectly nested loops case as conflicts between references to the same array can occur in both cases.

An important point to note is that the access matrices for these two references are completely different from each other. That is, our approach, under certain conditions, can optimize an array that has multiple (different–not necessarily uniformly generated) access matrices in the program. These conditions are discussed in Section 4.7.

4.5 Temporal Locality

A reference that has temporal locality in the innermost loop can be kept in a register (through scalarization) during the entire execution of the innermost loop. This can improve the performance substantially in many cases. It should be emphasized that the data layout transformations do not affect temporal locality of a reference directly [25]. However, if the trip count of the innermost loop is small, then the spatial locality in the second innermost loop may be important. Our approach takes this possibility into account. To see how this is done, consider again the previous loop nest, assuming that within the `k` loop there is an additional reference $U(i+j, j)$. The access matrix for this reference is

$$A_U = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

The last column of the matrix is zero, meaning that the reference exhibits temporal locality in the innermost loop. In this case, our method omits the zero column and considers the second column from right. Applying our method to this column, we select diagonal layout for this array. In general, the zero columns in the access matrices can be dropped from consideration.

4.6 Multiple Loop Nests

Notice that the ability to handle imperfectly nested loops within a unified framework also enables compiler to handle multiple loop nests in a single step. As proposed in [25], we can think of multiple loop nests as a single loop nest enclosed by an outermost loop with a trip count of one. That, of course, adds a zero column as a first column in the access matrices of all the references. But, since we are interested in the last columns of access matrices, it does not have any effect during the optimization process.

4.7 Condition for Conflict-Free Layout Optimization

Now, we focus on conditions that should hold for multiple references to the same array to be optimized in a conflict-free manner. In order to optimize spatial locality for the innermost loop, our approach focuses on the last columns of access matrices; thus, intuitively, if the Ker sets of the last columns of two references are *conformant,* (defined next) there will be no conflict in optimizing both.

Definition 6 *Let $\bar{x_1}, \bar{x_2}, \cdots, \bar{x_s}$ be s column vectors of the same size. The kernel sets of these vectors, namely, $Ker\{\bar{x_1}\}$, $Ker\{\bar{x_2}\}$, $\cdots$, $Ker\{\bar{x_s}\}$, are said to be conformant if for some i ($1 \leq i \leq s$), integer linear combinations of the basis vectors of $Ker\{\bar{x_i}\}$ can generate all vectors that belong to $Ker\{\bar{x_1}\}$, $Ker\{\bar{x_2}\}$, $\cdots$, $Ker\{\bar{x_s}\}$.*

Now we state the condition on conflict-free layouts as follows.

Lemma 3 *Suppose an array U is accessed by s different references with access matrices A_{U_1}, A_{U_2}, ..., A_{U_s}. Let a_{l_1}, a_{l_2}, ..., a_{l_s} be the last columns of these access matrices respectively. Then, these references can be optimized in a conflict-free manner if $Ker\{a_{l_1}\}$, $Ker\{a_{l_2}\}$, $\cdots$, $Ker\{a_{l_s}\}$ are conformant.*

Proof: Follows directly from the preceding discussion. □

The next theorem follows directly from Lemma (3).

Theorem 1 *Within a loop nest, if all references to a specific array are uniformly generated, then all the references can be optimized for spatial locality without any conflict.*

Proof: Trivial, as the UGRs have the same last columns in their access matrices. □

Let us now consider the following loop nest that accesses a two-dimensional array with four references.

```
do i = li, ui
  do j = lj, uj
    U(i+j,j) = U(j,i+j) + U(j,j) + U(i+j,2j)
  enddo
enddo
```

The access matrices are

$$A_{U_1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, A_{U_2} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, A_{U_3} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}, \text{ and } A_{U_4} = \begin{pmatrix} 1 & 1 \\ 0 & 2 \end{pmatrix}.$$

We note that $Ker\{a_{l_1}\}$, $Ker\{a_{l_2}\}$ and $Ker\{a_{l_3}\}$ are conformant but not $Ker\{a_{l_4}\}$. Therefore, the first three references can be optimized without conflict whereas there is a conflict with the fourth reference. In such a case, our method favors the largest subset of conflict-free references [33, 36] and decides a diagonal layout with the hyperplane $(1, -1)$ instead of $(2, -1)$. In the general case, however, whenever conflicting references occur in different nesting levels, a *conflict resolution* scheme is needed.

Our conflict resolution scheme is based on the *weight* of references. Essentially, the weight of a reference is the number of times it is accessed. This number can be estimated by multiplying the trip counts (the number of iterations) of the loops which enclose that reference. If the trip counts are not available at compile-time, we use profiling to get this information. In practice, average trip count estimations are sufficient for a majority of applications.

After the weight of each reference is obtained, the references are divided into groups such that two references belong to the same group if they are conformant in the sense defined above. It is easy to see that two references are conformant if *canonical forms* of the last columns of their access matrices are equal. The canonical form of a column vector $\bar{x} = (x_1, x_2, \cdots, x_e)^T$ is $Canonical(\bar{x}) = (x_1/g, x_2/g, \cdots, x_e/g)^T$ where $g = \gcd\{x_1, x_2, \cdots, x_e\}$.

Thus, two references R_1 and R_2 belong to the same conformity group if the last columns of their access matrices, $\bar{x}_{R_1}$ and $\bar{x}_{R_2}$ satisfy

$$Canonical(\bar{x}_{R_1}) = Canonical(\bar{x}_{R_2}).$$

After determining the conformity groups, the compiler computes the *weight* of each conformity group. The weight of a conformity group is sum of the weights of the references it contains. Assuming that a conformity group C_i contains references $R_1, R_2, \cdots$ and R_f, the weight of C_i can be computed as

$$weight(C_i) = \sum_{j=1}^{f} weight(R_j) = \sum_{j=1}^{f} \prod_{l} \textit{trip count(l)},$$

where l iterates over the loops that enclose R_j.

Once the weights of the conformity groups are computed, our approach chooses a reference from a conformity group with the largest weight as *representative reference*. Then the compiler proceeds to optimize locality for this reference which in turn would satisfy the majority of references. Notice that all the references in a conformity group can be assumed to be the same as far as the locality is concerned, because we are only interested in the last columns.

In our experiments, this conflict resolution scheme was needed only in a single array in a single program (`btrix`). Although our approach obviously cannot be optimal for every case, we expect that in practice it will be successful.

Contrast the last example with the following that also contains a non-unit stride access.

```
do i = li, ui
  do j = lj, uj
    U(i,j) = U(i,2j) + U(i,i+j)
  enddo
enddo
```

The access matrices are

$$A_{U_1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, A_{U_2} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}, \text{ and } A_{U_3} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

Since, $Ker\{a_{l_1}\}$, $Ker\{a_{l_2}\}$ and $Ker\{a_{l_3}\}$ are conformant (i.e., they are generated by the vector $(1,0)^T$), the array U can be stored in memory as row-major without any conflict among the references. Notice that while the non-unit stride causes a conflict in the previous example, in this example it does not cause a conflict; rather it causes underutilization of cache lines (for the second reference). Our framework, with appropriate modifications, can be extended to handle non-unit stride accesses as well by shrinking arrays whenever possible.

5 Multiprocessors

Our data layout optimization technique works well in a multiprocessor environment as well for the following reasons:

- Optimizing compilers are in general successful in parallelizing the outermost loops [40].
- Our data layout optimization framework generates a code such that (if possible) only the innermost loops carry spatial reuse.

To see why these help in a multiprocessor environment, it is sufficient to note that parallelizing a loop that carries spatial locality is one of the main reasons for false sharing, a phenomenon that occurs when two or more processors access logically separate data placed on the same cache line [39]. Since our framework causes spatial locality to be carried by the innermost loops, in most cases, the compiler can safely parallelize (provided the dependences allow it) the outermost loops without an apparent danger of false sharing.

In those situations where the compiler is able to parallelize only the innermost loops, the performance of data layout transformations in multiprocessors may be rather poor. Because, in those cases, the parallelized loops will also carry spatial reuse.

In this section, we present a locality enhancing approach that takes into account parallelism as well. Our approach is based on the spatial reuse vectors which are explained in Section 5.1. In Section 5.2, on the other hand, we present sketch of an algorithm that can optimize spatial locality and at the same time reduce the false sharing.

5.1 Spatial Reuse Vectors under Different Layouts

Let us first concentrate on how to optimize a given loop nest for a target spatial reuse vector. Consider the following two-deep loop nest.

```
do i = li, ui
  do j = lj, uj
    U(j+1,i-1) = V(i-1,j+1) + W(j-1,i+j+1)
  enddo
enddo
```

Assuming a column-major memory layout for all arrays, a spatial reuse occurs when the same column of an array is accessed more than once. 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 $\bar{I}$ and $\bar{J}$ access elements residing in the same column, they should satisfy the condition $A_{U_s}\bar{I} = A_{U_s}\bar{J}$, where A_{U_s} is the matrix A_U (for an array U) with the first row deleted [43]. From this condition, since A_{U_s} represents a linear mapping,

$$A_{U_s}(\bar{J} - \bar{I}) = \bar{0} \Rightarrow \bar{J} - \bar{I} \in Ker\{A_{U_s}\} \Rightarrow r_{\bar{U}} \in Ker\{A_{U_s}\}$$

where $\bar{r_U} = \bar{J} - \bar{I}$.

In that case, $\bar{r_U}$ is termed as the spatial reuse vector and $Ker\{A_{U_s}\}$ is referred as spatial reuse vector space [43]. The loop that corresponds to the first non-zero element of the basis vector of $Ker\{A_{U_s}\}$ is said to carry spatial reuse. As an example, 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 $\bar{r_U} = (0,1)^T$, $\bar{r_V} = (1,0)^T$, and $\bar{r_W} = (1,-1)^T$. Notice that the best possible spatial reuse vector is $(0,0,\cdots,0,1)^T$ which indicates that the spatial locality is carried (and exploited) by the innermost loop.

Although the spatial reuse vectors 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. What we need is a new definition of reuse vector under different memory layouts.

Lemma 4 *Let $\bar{g}$ represents a memory layout for an array U, and $\bar{r_U}$ the spatial reuse vector associated with a reference represented by the access matrix A_U. Then, the following equality holds between $\bar{g}$, $\bar{r_U}$ and A_U:*

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

Proof: Let $\bar{I}$ and $\bar{J}$ be two iteration vectors, and $\bar{r_U} = \bar{J} - \bar{I}$. The data elements accessed by these vectors using A_U are $A_U \bar{I}$ and $A_U \bar{J}$ respectively. These two elements have spatial locality if $g^T A_U \bar{I} = g^T A_U \bar{J}$, i.e., $g^T A_U(\bar{J} - \bar{I}) = 0 \Longrightarrow g^T A_U \bar{r_U} = 0$ □

If the memory layout of an array is represented by a matrix L (as in three- or higher dimensional cases), then 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. Assume now that we would like to restructure the data layout to obtain a target reuse vector $\bar{r_U}'$. In that case

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

This means that the basis vector of $Ker\{A_U \bar{r_U}'\}$ can be the desired hyperplane vector that represents the memory layout to obtain $\bar{r_U}'$ as the spatial reuse vector. This last equation can be used to select a memory layout for a given spatial reuse vector.

5.2 Sketch of an Algorithm for Optimizing Locality and Reducing False Sharing

For shared-memory multiprocessors, 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 that carries spatial reuse should not be parallelized [26]. 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 [17].

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 $\bar{p}$ where $p(i)$ (the i^{th} element) is one if the loop i is parallelized otherwise it is zero.

```
FOR EACH array U ∈ 𝒰
   compute the access matrix A_U representing U
   i = n
   WHILE (p(i) = 1)
      i = i − 1
   END WHILE
   r̄ = ē_i
   compute 𝒩 = Ker{(A_U r̄)^T}
   use the elements of the basis set of 𝒩 as rows of the layout constraint matrix for U
END FOR EACH
```

Figure 2: Algorithm for optimizing spatial locality.

The memory layout determination algorithm is given in Figure 2. 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 $\bar{e_i}$ 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 the conflict resolution scheme mentioned earlier should be applied to find a representative access matrix. 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 $\bar{e_n}$, ..., $\bar{e_2}$, $\bar{e_1}$. A vector is rejected as target spatial reuse vector if the associated loop is parallelized. Otherwise $\bar{e_i}$ 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 (Ker set) and a set of basis vectors for it. These basis vectors represent 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 also be emphasized that although we present our technique for a single nest, it can be extended for multiple nests by enclosing the nests in question with an outermost loop that iterates only once. This is possible because data transformations are applicable to imperfectly nested loops as well. In that case, the form of the parallelization information available to our algorithm should also be modified.

6 Determination of Data Transformation Matrix and Code Generation

In this section, we show how to determine a suitable transformation matrix for each target layout. We propose a strategy to find the necessary data transformation matrix in order to obtain the ‘effect’ of the desired (optimal) layout taking into account the default (canonical) layout adopted by the compiler. Notice that the notion of data transformation matrix is *different* from that of layout constraint matrix. The previous research [31] investigated the different types of non-singular data space transformation matrices and their effects on the behavior of programs.

A data transformation M_U for an array U is applied in two steps:

- First, the access matrix (resp. offset vector) is transformed to $M_U A_U$ (resp. $M_U \bar{o}$).
- Second, the array bounds (i.e., array declarations) are changed accordingly.

Let us first focus on the first step. Our transformation framework uses the concept of layout constraint matrix, introduced earlier. Assume that for an array U, $L_{U_{\text{default}}}$ is the default layout (which we assume for now, without loss of generality, column-major following the Fortran convention); and $L_{U_{\text{desired}}}$ be the optimized layout. It is easy to see that, a data transformation matrix M_U can convert the default layout to the optimized layout if

$$L_{U_{\text{default}}} M_U = L_{U_{\text{desired}}}. \tag{5}$$

When this equation is solved for the elements of M_U, some of the elements of M_U (or at least some relations between them) will be determined. The remaining elements can be filled in any way provided that the resulting matrix will be non-singular.

Consider the following program discussed earlier, assuming that the default layout is column-major; i.e., $L_{U_{\text{default}}} = (0, 1)$.

```
do i = li, ui
  do j = lj, uj
    U(i,j) = V(j,i) + W(i+j,i) + X(i+j,j) + Y(n-j,i+j)
  enddo
enddo
```

From the previous section, the desired layout matrices are

$$L_{U_{\text{desired}}} = (1, 0), L_{V_{\text{desired}}} = (0, 1), L_{W_{\text{desired}}} = (0, 1), L_{X_{\text{desired}}} = (1, -1), \text{ and } L_{Y_{\text{desired}}} = (1, 1).$$

Using equation (5), we derive:

$$(0,1)M_U = (1,0) \Longrightarrow M_U = \begin{pmatrix} \times & \times \\ 1 & 0 \end{pmatrix} \qquad (0,1)M_V = (0,1) \Longrightarrow M_V = \begin{pmatrix} \times & \times \\ 0 & 1 \end{pmatrix}$$

$$(0,1)M_W = (0,1) \Longrightarrow M_W = \begin{pmatrix} \times & \times \\ 0 & 1 \end{pmatrix} \qquad (0,1)M_X = (1,-1) \Longrightarrow M_X = \begin{pmatrix} \times & \times \\ 1 & -1 \end{pmatrix}$$

$$(0,1)M_Y = (1,1) \Longrightarrow M_Y = \begin{pmatrix} \times & \times \\ 1 & 1 \end{pmatrix}$$

where $\times$ stands for an unknown entry. These matrices should be completed such that they should be nonsingular. Fortunately, the completion algorithms offered by [26] can be used for this purpose. An example solution is

$$M_U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, M_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, M_W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, M_X = \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix}, \text{ and } M_Y = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

From these matrices, compiler obtains the transformed access matrices as follows:

$$M_U A_U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad M_V A_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$M_W A_W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \qquad M_X A_X = \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$M_Y A_Y = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

The offset vectors are transformed in a similar manner. The transformed loop nest is as follows.

```
do i = li, ui
  do j = lj, uj
    U(j,i) = V(j,i) + W(i+j,i) + X(i+j,i) + Y(n-j,i+n)
  enddo
enddo
```

The order of the loops and the loop bounds themselves are unchanged; but the array declarations should be changed. We will come to this issue shortly. Notice that, all of the transformed references exhibit good spatial locality (therefore, the loop nest exhibits perfect spatial locality), for the column major layouts.

We can also transform the same original loop nest assuming that the default memory layout is row-major. In this case, the default layout can be expressed as hyperplane vector $(1, 0)$. Using equation (5) again, we can obtain the following transformed nest.

```
do i = li, ui
  do j = lj, uj
    U(i,j) = V(i,j) + W(i,i+j) + X(i,i+j) + Y(i+2n,n-j)
  enddo
enddo
```

Notice that the spatial locality is perfect for row-major layouts. And finally, we can optimize the same loop assuming that the default layout is diagonal; that is the hyperplane vector is $(1, -1)$. The resulting code is as follows:

```
do i = li, ui
  do j = lj, uj
    U(i+j,j) = V(i+j,j) + W(2i+j,i+j) + X(i+j,j) + Y(2i+j+n,i+j)
  enddo
enddo
```

We note that the spatial locality is perfect for diagonal layouts.

Let us now focus on the problem of modifying the array declaration statements. Consider the following example:

```
real U(n/2,n,n)

  do i = 1, n
    do j = 1, n/2
      do k = 1, n/2
        ...  U(k,j+k,i) ...
      enddo
    enddo
  enddo
```

Our approach uses the method of extreme values of affine functions first used by Banerjee [4, 44] in dependence testing. Given an affine function of a number of variables and inequalities that represent the bounds for the variables, the extreme values method finds the maximum and minimum values of the affine function in the bounded region. The method applies to non-rectilinear regions as well (see [4, 44]). In this method, the computed extreme values may not be tight as could have been if the Fourier-Motzkin elimination [37] had been used. However, for many programs, applying Fourier-Motzkin elimination to obtain tight bounds may be too expensive. The extreme-values method can be best explained using an example. Consider the affine function $f(x, y) = 3x - 2y + 1$ with the region $\{1 \leq x \leq 10$ and $x + 1 \leq y \leq 30 - x\}$.

For the upper bound:

$$\begin{aligned} f(x,y) &\leq 3x - 2y + 1 \\ f(x,y) &\leq 3x - 2(x+1) + 1 \\ f(x,y) &\leq x - 1 \\ f(x,y) &\leq 9 \end{aligned}$$

For the lower bound:

$$\begin{aligned} f(x,y) &\geq 3x - 2y + 1 \\ f(x,y) &\geq 3x - 2(30-x) + 1 \\ f(x,y) &\geq 5x - 59 \\ f(x,y) &\geq -54 \end{aligned}$$

Therefore, the upper bound of $f(x, y)$ is 9 and the lower bound is -54. Returning to our example, a suitable transformation matrix is $\begin{pmatrix} 0 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$. What this means is that the subscript expressions should be changed from `(k,j+k,i)` to `(j+k,-j,i)`. The resultant affine functions in this example are $f_1(i, j, k) = j + k$, $f_2(i, j, k) = -j$ and $f_3(i, j, k) = i$. For f_3, the lower and upper bounds are 1 and n respectively. For f_2, on the other hand, the lower and upper bounds are $-n/2$ and -1 respectively. For f_1, we calculate the bounds as follows.

For the upper bound:

$$\begin{aligned} f_1(i,j,k) &\leq j + k \\ f_1(i,j,k) &\leq j + (n/2) \\ f_1(i,j,k) &\leq n \end{aligned}$$

For the lower bound:

$$\begin{aligned} f_1(i,j,k) &\geq j + k \\ f_1(i,j,k) &\geq j + 1 \\ f_1(i,j,k) &\geq 2 \end{aligned}$$

Therefore, the upper bound of f_1 is n and the lower bound is 2. The transformed program is as shown next.

```
real U(2:n,-n/2:-1,1:n)

  do i = 1, n
    do j = 1, n/2
      do k = 1, n/2
        ...  U(j+k,-j,i) ...
      enddo
    enddo
  enddo
```

As a final point, notice that our approach is able to find a data transformation matrix to convert one memory layout into another. We believe that such a technique will also be quite useful in a compilation environment that considers dynamic layout transformations between loop nest boundaries. In such an environment, an optimizing compiler may have to solve the problem of, say, converting a diagonal layout to row-major layout or a column-major layout to an anti-diagonal layout etc. It is easy to see that our framework can readily handle those and the similar cases.

7 Experimental Results

In this section, we present performance results for both uniprocessor and multiprocessor cases. Our performance results include uniprocessor cache miss rates and uniprocessor/multiprocessor execution times.

7.1 Experimental Suite

We present performance results for eight programs: `matmult` is a matrix-multiplication routine with two versions (with and without loop unrolling)[1]; `syr2k` is banded matrix update routine from the BLAS library [9]; `adi` is from Livermore kernels; `btrix`, `vpenta` and `cholesky` are from Spec92/NASA benchmark suite; and `transpose` is a matrix transpose program from NWChem [15], a software package for computational chemistry.

We conduct experiments with C versions of these programs. For each program in the suite, we experiment with four different versions:

- `col`: original program with column-major layout for all arrays;
- `row`: original program with row-major layout for all arrays;
- `lopt`: a version optimized by loop transformation techniques; (and)
- `dopt`: a version optimized by data layout transformations studied in this paper.

The `lopt` versions are obtained by either applying Li's locality optimization approach [26] or allowing the native compiler to derive the best order (whichever performs best).

In the experiments on multiple nodes, where possible, the coarsest granularity of parallelism is exploited for all versions. In fact, for the `row`, `col` and `dopt` versions exactly the same set of loops are parallelized; therefore the degree of parallelism is the same. For all programs, the degree of parallelism of the `lopt` version is either better than or at least as good as the other versions.

7.2 Platforms

We report execution times in seconds for up to 8 processors on a Convex Exemplar SPP-2000 and an SGI Origin 2000, two distributed-shared-memory machines. The 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 may have up to 16 processors, 16GB of memory and a CTI cache of size 256MB. The programs are compiled using the native C compiler on the Exemplar with the `-O3` option. The timings are taken using the CXpa performance analyzer. The SGI Origin 2000 uses 195MHz R10000 processors with 32KB L1 data cache and 4MB L2 unified cache. The programs for the SGI Origin are compiled using the native C compiler with the `-O2` option. The timings are taken using the *gettimeofday* routine.

[1] The unrolled version is called `matmult/u`.

Table 3: Uniprocessor miss rates for the programs in our experiment suite [`bs` = block (line) and `cs` = cache size]. [The problem sizes (in double-precision) are as follows: `matmult` and `matmult/u` – 1024×1024 matrices; `syr2k` – 1024×1024 matrices and $b = 300$; `vpenta` – $4 \times 720 \times 720$ 3D arrays and 720×720 2D arrays; `cholsky` – the size parameters are set to 2500; `btrix` – the size parameters are set to 150; `adi` – $1000 \times 1000 \times 3$ arrays; and `transpose` – two 2048×2048 matrices. The programs from Spec92 and the `adi` and `transpose` codes have an outer timing loop. The same sizes are used in run-time experiments as well].

Program	bs	cs = 8K				cs = 32K				cs = 128K				cs = 512K			
		col	row	lopt	dopt	col	row	lopt	dopt	col	row	lopt	dopt	col	row	lopt	dopt
matmult	16	.441	.317	.226	.223	.394	.271	.156	.154	.383	.259	.138	.136	.012	.011	.013	.011
matmult	32	.382	.320	.182	.179	.336	.274	.100	.098	.324	.262	.080	.078	.014	.012	.014	.012
matmult	64	.357	.327	.165	.162	.312	.281	.078	.076	.300	.270	.056	.054	.020	.019	.021	.019
matmult	128	.354	.339	.167	.164	.310	.296	.078	.076	.299	.285	.056	.054	.033	.034	.035	.033
matmult/u	16	.255	.317	.287	.177	.185	.130	.101	.072	.167	.084	.054	.045	.013	.014	.016	.012
matmult/u	32	.245	.321	.274	.158	.175	.133	.088	.058	.158	.087	.042	.033	.016	.015	.017	.013
matmult/u	64	.219	.327	.274	.155	.144	.141	.071	.057	.125	.094	.041	.032	.022	.022	.023	.019
matmult/u	128	.217	.341	.283	.166	.141	.155	.099	.068	.121	.109	.052	.043	.035	.037	.038	.034
syr2k	16	.664	.417	.365	.206	.658	.291	.316	.179	.527	.173	.281	.157	.100	.041	.131	.013
syr2k	32	.662	.542	.187	.109	.655	.353	.160	.092	.514	.105	.142	.080	.058	.024	.067	.007
syr2k	64	.660	.628	.101	.066	.650	.416	.082	.049	.501	.091	.073	.043	.044	.019	.034	.004
syr2k	128	.659	.646	.072	.057	.640	.525	.043	.030	.492	.246	.038	.025	.051	.023	.018	.003
vpenta	16	.652	.502	.502	.457	.652	.498	.498	.457	.435	.403	.403	.311	.218	.272	.272	.207
vpenta	32	.543	.321	.321	.251	.544	.317	.317	.250	.475	.294	.294	.177	.133	.166	.166	.131
vpenta	64	.489	.236	.236	.151	.489	.226	.226	.148	.487	.217	.217	.111	.236	.113	.113	.072
vpenta	128	.462	.227	.227	.139	.462	.181	.181	.098	.487	.177	.177	.078	.433	.087	.087	.043
cholesky	16	.615	.730	.206	.352	.481	.575	.183	.285	.263	.326	.148	.192	.167	.189	.082	.133
cholesky	32	.566	.706	.158	.177	.484	.658	.104	.143	.340	.372	.074	.096	.114	.168	.041	.066
cholesky	64	.561	.659	.169	.089	.529	.658	.098	.072	.362	.411	.038	.048	.144	.200	.021	.033
cholesky	128	.411	.593	.190	.045	.349	.590	.096	.036	.237	.445	.020	.024	.099	.304	.011	.017
btrix	16	.228	.221	.223	.223	.196	.180	.196	.182	.137	.099	.137	.120	.118	.075	.118	.091
btrix	32	.142	.152	.142	.135	.117	.109	.117	.100	.079	.053	.079	.064	.066	.039	.066	.048
btrix	64	.108	.139	.103	.099	.075	.085	.075	.061	.049	.040	.049	.036	.039	.020	.039	.026
btrix	128	.118	.243	.110	.110	.054	.096	.054	.043	.033	.040	.033	.022	.024	.017	.024	.014
adi	16	.493	.466	.350	.388	.465	.349	.234	.267	.310	.310	.156	.178	.282	.301	.156	.162
adi	32	.375	.479	.319	.343	.312	.330	.119	.135	.202	.286	.079	.090	.182	.280	.078	.082
adi	64	.272	.475	.322	.346	.160	.367	.132	.134	.102	.306	.040	.047	.091	.302	.039	.043
adi	128	.356	.555	.412	.452	.186	.465	.249	.278	.053	.311	.022	.025	.046	.307	.020	.023
transpose	16	.625	.625	.625	.501	.375	.375	.375	.250	.281	.281	.281	.250	.258	.258	.258	.250
transpose	32	.813	.813	.813	.753	.344	.344	.344	.125	.180	.180	.180	.125	.139	.139	.139	.125
transpose	64	.906	.906	.906	.874	.453	.453	.453	.063	.130	.130	.130	.063	.080	.080	.080	.063
transpose	128	.953	.953	.953	.938	.445	.445	.445	.031	.104	.104	.104	.031	.050	.050	.050	.031

7.3 Performance

Table 3 shows the single processor miss rates obtained using an enhanced version of DineroIII [14] for different direct-mapped cache configurations. These results show that our technique is quite successful in reducing miss rates.

The results from the Exemplar are given in Figure 3. For `syr2k` and `transpose`, `dopt` is the winner. The `syr2k` code accesses two out of three arrays anti-diagonally, so an anti-diagonal data layout improves the locality without making the subscript functions too complex. The `lopt` version (from [26]), on the other hand, improves spatial locality for two arrays, but converts temporal locality for the third array into spatial locality; in addition, it generates complex loop bounds and subscript expressions. The `transpose` code accesses two arrays with conflicting patterns (row-major and column-major). Therefore, `lopt` cannot optimize this program.

For `matmult`, the performances of `lopt` (from [26]) and `dopt` are similar. When unrolling turned off, for 1 and 2 processor cases, `lopt` is better while for all other cases `dopt` performs better. When the outermost two loops are unrolled with a factor of 4, `dopt` outperforms `lopt` in Exemplar except for the uniprocessor case. The `matmult` code is an example of a group of programs where loop transformations and data transformations generate competitive codes from the locality point of view.

The `adi` benchmark is a program that consists of two nests. For uniprocessor case, `dopt` is better than `lopt`. Beyond a single processor, `lopt` is better, though the performance of `dopt` is close. Notice that for all programs explained so far `dopt` outperforms the versions with fixed layouts (`col` and `row`), indicating the importance of data layout optimizations.

The benchmarks `vpenta`, `cholesky` and `btrix` are relatively large programs consisting several nests. In Convex Exemplar SPP-2000, `dopt` successfully optimizes `vpenta`. The `lopt` version outperforms `dopt` beyond 4 processors, simply because `lopt` also exploits the parallelism better, an improvement that cannot be obtained using data transformations alone. In `cholesky`, the `lopt` version is the clear winner as it exploits both parallelism and locality by several loop transformations including loop permutation *and* loop fission. For this example, `dopt` also introduces some false sharing which prevents the success of layout optimizations. For `btrix`, on the other hand, the performances of `dopt` and `lopt` are similar.

The results on the SGI Origin 2000 are given in Figure 4. As on the Convex Exemplar SPP-2000, for `syr2k` and `transpose`, `dopt` is the winner. The `dopt` version is also winner for `vpenta` up to 4 processors. For `matmult`, `lopt` outperforms `dopt`, though the performances are very similar. For the `cholesky` program, the `lopt` version is the clear winner, mostly because of improved parallelism. For `adi`, the `lopt` version is better than `dopt`. For `btrix`, on the other hand, up to 4 processors, `dopt` performs better than `lopt`; beyond 4 processors, however, the false sharing dominates and `dopt` performs poorly.

7.4 Observations

From our experiments, we observe the following:

- Data layout transformations are very effective in optimizing cache locality for uniprocessors. Specifically for five out of eight cases on the Convex Exemplar SPP-2000 as well as on the SGI Origin 2000, the `dopt` version outperforms the other versions (including `lopt`) in a single node.

- Our transformations can optimize some programs such as `syr2k` and `transpose` that cannot be fully optimized by loop transformations.

- For multiple loop nests, our framework is quite successful in detecting the optimal layouts due to the fact that (except `btrix`) there was no conflict between the optimized layout requirements of individual nests.

- Although in general data layout optimizations can be considered successful for multiprocessor case, the performance may be rather poor in cases where outermost loop parallelism cannot be obtained, Also, in some cases, loop transformations may also improve parallelism; thus, may have additional advantage over data transformations (as in `cholesky`).

- The relative performances of different versions across two platforms are different. We attribute this to the different cache sizes, page sizes, page allocation policies and memory organizations of the Exemplar and the Origin.

8 Interaction Between Data Distribution and Data Layout

On distributed shared memory (DSM) machines, *cache locality* improvements (through data layout transformations) and *memory locality* improvements (through data distribution) are complementary. While good cache locality optimization combined with a good page management scheme (such as first-touch (FT) policy with page migration [24, 41]) can effectively ensure low memory access costs, merely distributing the data across the memories of the processors in a best possible way may not necessarily ensure good cache locality.

We would like to delve a bit more into this interplay between cache locality optimization, page management policies and data distribution on DSM machines. A detailed study of such interactions along with how a compiler can optimize codes for good overall performance, however, is beyond the scope of this paper. In the Convex Exemplar SPP-2000, the page placement policy is round robin (RR) where the pages are allocated in a round-robin fashion across processors. In the experiments presented in the previous section, we used this policy[2]. The SGI Origin 2000, on the other hand, uses FT as the default policy but provides a couple of environment variables and system calls to select RR policy as well as to enable or disable page migration. In the experiments on the SGI Origin, we employed the default FT page allocation policy. We have chosen the SGI Origin as our example target in this section for investigating the interplay between cache locality and memory locality since this machine not only supports user controllable page management strategies but also HPF like data distribution compiler directives [6]. A typical directive is of the form `distribute` $U(\langle dist_1\rangle, \langle dist_2\rangle, \cdots, \langle dist_m\rangle)$, where U is an m-dimensional array and $\langle dist_i\rangle$ may be one of *block*, *cyclic*, *cyclic*$\langle$*expr*$\rangle$ or $*$, with the same meaning as in HPF. Explicit data distribution is allowed to be *regular* or *reshaped* [6]. In a regular distribution, each processors' portion is mapped onto pages from its own memory. In a reshaped distribution, the layout of data within the virtual address space is changed to make the data accessed by each processor local. For example, column-wise distribution of a very large array which is stored row-major in memory may require reshaped distribution.

In theory, a compiler can use automatic static (e.g., [13, 35, 23, 11, 38, 27, 7]) or dynamic (e.g., [12, 32, 3]) data distribution methods developed for message-passing architectures or non-uniform memory access (NUMA) machines; once the optimal data distributions are determined, the appropriate distribution directives can be inserted in the code.

We considered the C versions of `transpose` and `matmult` as case studies. We chose these programs as, given an outermost loop parallelism, it is easy to find good data distributions for them using the techniques in the literature (e.g., [13]). Figures 5(a) and (b) show the breakdown of misses into three classes, primary cache (PC), secondary cache (SC), and translation look aside buffer (TLB) misses, for the `transpose` code on 8 processors using 1000×1000 and 5000×5000 matrices, respectively. There is an outermost timing loop which iterates 10 times. The first four bars in each figure correspond to (cache) unoptimized programs whereas the remaining four bars correspond to cache-optimized programs using the approach explained in this paper. FT, RR, RG and RS correspond to the program versions with the first-touch, round-robin, regular distribution and reshaped distribution respectively. In the case of regular distribution, one of the arrays is distributed row-wise whereas the other distributed column-wise to eliminate communication. But, since the memory layout is *row-major*, the column-wise distributed layout will cause memory

[2] Although under the SPP-UX operating system there is a utility called 'mpa' which can be used to set some of the global attributes (program wide policies such as which memory class different data structure types map to etc.), using this utility appropriately does not seem trivial.

locality problems unless it is reshaped.

From Figure 5(a) we can see that if no layout optimization is performed, the FT, RR and RG versions exhibit similar behavior. The RS version, on the other hand, reduces the TLB misses significantly, and reduces some of the PC misses. When we apply layout optimizations, there is a huge reduction in PC and TLB misses for almost all versions. The RG and RS versions additionally reduce the number of SC misses. We note that the performance of the program under layout optimization becomes almost independent of data distribution. When the input size is increased, if no layout optimization is done, the RS version reduces the TLB misses but increases the PC misses as the data size is much larger compared to the size of the first-level cache. With layout optimizations, all types of misses are reduced significantly. But since the R10000 is a complex super-scalar processor, a lot of misses can be hidden (concurrently serviced with the ongoing computation); therefore, the ultimate criterion is the execution times; the execution times are shown in Figures 6(a) and (b) for 8 processors. It is clear that in the optimized programs the RG and the RS versions behave similarly and result in the best performance. It should be emphasized that the in the Origin that we used there is approximately a 1:2 ratio between the local and remote memory latencies. This fact makes the FC and SC misses as important as the TLB misses.

The miss decomposition for our second example, `matmult`, is shown in Figure 7(a) for 2500×2500 matrices on 8 processors. It is interesting to note that the (cache) unoptimized RS version and the cache-optimized versions show similar performance. The execution times in Figure 7(b) also follow the same trend showing that the best version is the layout optimized RG code.

It is important to understand the reason why these two examples behave differently. In the `transpose` code, there are accesses to two arrays with different optimal layouts. The best data decomposition is to distribute one of the arrays in $(block, *)$ fashion and the other in $(*, block)$ fashion. But, even after optimal data distribution, we still need data layout optimizations to optimize all references for perfect spatial locality. That is why the memory optimized version (without cache optimizations) does not exhibit good cache locality. In `matmult`, on the other hand, after the reshape, the loop transformation framework of the SGI Origin is able to optimize all the transformed references for locality. In that case, both the pure memory-optimized version and the pure cache-optimized version result in competitive codes.

In conclusion, optimizing compilers should attempt to improve both cache and memory locality in a unified framework. As far as the method presented in this paper is concerned, under a fixed data distribution (or page allocation) strategy, we obtain improvements using data layout transformations over the unoptimized programs.

9 Related Work

Loop transformations have been used for optimizing cache locality in several papers [26, 43, 30]. Results have shown that on several architectures the speedups achieved by loop transformations alone can be quite large. McKinley et al. [30] offer a unified optimization technique consisting of loop permutation, loop fusion and loop distribution. By considering iteration space transformations only, they obtain impressive speedups for several programs. Wolf and Lam [43] propose a data locality optimization algorithm based on a mathematical description of reuse. They identify and quantify reuse within an iteration space. They use vector spaces to represent the directions where reuse occurs and define different types of reuses found in dense matrix programs. Their approach is sort of exhaustive in the sense that they try all possible subsets of the loops in the nest, and (if necessary) by applying unimodular transformations they bring the subset with the best potential reuse into the innermost positions. In comparison, Li [26] describes a data reuse model and a compiler algorithm called height reduction to improve cache locality. He discusses the concept of a data reuse vector and defines its height as the number of dimensions from the first non-zero entry to the last entry. The non-zero entries of a reuse vector indicate that there are reuses carried by the corresponding loops. The algorithm assigns priorities to reuse vectors depending on the number of times they occur, and reduces the height of a global reuse matrix starting from the reuse vector of highest priority. As compared with Wolf and Lam's approach, Li's approach is faster and can keep reuse vector information more precisely. None of [43], [26], [30] considers data space transformations. In this paper, we show that data space transformations can also make difference on the locality properties of the programs. Moreover, for some cases they are applicable where iteration space transformations are not.

Only a few papers have considered data transformations to optimize locality. O'Boyle and Knijnenburg [31] focus on restructuring the code given a data transformation matrix; though they show their method can be used for optimizing spatial locality. In comparison, we concentrate more on the problem of determining suitable layouts, derive a simple technique that can be applicable to multiple nests as well. Anderson, Amarasinghe and Lam [2] propose a data transformation technique for distributed shared memory machines. By using two types of data transformations (strip-mining and permutation), they try to make the data accessed by the same processor contiguous in the shared address space. Their algorithm inherits parallelism decisions made by a previous phase of the SUIF compiler [42]. While they restrict themselves to strip-mining and permutation only, we consider all types of layout transformations expressible by hyperplanes. This allows compiler, for instance, to optimize the `syr2k` code. Ju and Dietz [18] 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. Cierniak and Li [8] present a unified approach to optimize locality. Their approach employs both data space and iteration space transformations. The notion of 'stride vector' is introduced and an optimization strategy is developed for obtaining the desired mapping vectors and transformation matrix. The main drawbacks of this approach are as follows: (1) Their method conducts an exhaustive search in a constrained search space for the best transformation; (2) It restricts the search space by enforcing the entries of the iteration space transformation matrix to be 1 or 0; (3) The memory mappings considered are a subset of possible mappings and do not contain diagonal (or skewed) mappings; and (4) Their approach depends on array bounds; this in turn, implies either availability of these bounds during compilation or manipulation of symbolic expressions. Leung and Zahorjan [25] present an array restructuring framework to optimize locality. 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 expensive Fourier-Motzkin elimination.

Third, their code generation algorithm is also different from ours. Previous work of the authors [21, 22] considers only dimension re-indexing and and like [8] uses a sort of exhaustive search to detect array layouts. In contrast to [21, 22], and [8], the approach explained in this paper considers a much larger search space for memory layouts, and finds the optimal layouts in a single step.

Finally, as mentioned earlier, there is a huge body of work on automatic data distribution on distributed memory machines (e.g., [7, 3, 11, 12, 13, 23, 27, 32, 38, 36]). The interplay between that work and ours have been discussed earlier in the paper.

10 Summary

In this paper, we presented an approach based on the theory of hyperplanes and and the linear algebra framework used by parallelizing compilers for optimizing memory layouts of arrays. Our approach divides the layout optimization problem into two subproblems: (1) detecting optimal layouts for each array and (2) implementing optimal layouts within a compiler that has a default layout for all arrays. We mainly concentrated on the first subproblem, and showed that our technique can work with different compilers with different default layouts. We also presented some important results about data layout optimizations and showed that for given a loop nest, data layout transformations can be used to obtain perfect spatial locality, provided that some conditions are satisfied. Experimental results on a Convex Exemplar SPP-2000 and an SGI Origin 2000 indicate that our framework can find opportunities for optimizing spatial locality without changing the access pattern of the loops. Our technique can be applied to any architecture with a memory hierarchy including uniprocessor machines. We are working on integrating loop and data transformations in a unified framework based on hyperplanes.

References

[1] B. Appelbe and B. Lakshmanan. Optimizing parallel programs using affinity regions. In *Proc. 1993 International Conf. Parallel Processing,* pages 246–249, St. Charles, IL, August 1993.

[2] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th ACM SIGPLAN Symp. Principles & Practice of Parallel Programming*, pages 166–178, Santa Barbara, CA, July 1995.

[3] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. SIGPLAN Conf. Programming Language Design & Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[4] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1988.

[5] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, A. Nicolau et al. (Eds.), MIT Press, 1991.

[6] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. Anderson. Data-distribution support on distributed-shared memory multiprocessors. *Proc. SIGPLAN Conf. Programming Language Design & Implementation*, Las Vegas, NV, pages 334–345, 1997.

[7] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Optimal Evaluation of Array Expressions on Massively Parallel Machines. *ACM Transactions on Programming Languages and Systems,* 17(1):123-156, January 1995.

[8] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. *Proc. SIGPLAN Conf. Programming Language Design & Implementation*, La Jolla, CA, pages 205–217, June 1995.

[9] 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.

[10] 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(5):587–616, October 1988.

[11] J. Garcia, E. Ayguade, and J. Labarta. A novel approach towards automatic data distribution. In *Proc. Supercomputing'95*, San Diego, December 1995.

[12] J. Garcia, E. Ayguade, and J. Labarta. Dynamic data distribution with control flow analysis. In *Proc. Supercomputing'96*, Pittsburgh, November 1996.

[13] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[14] M. Hill and A. Smith. Evaluating associativity in CPU caches. *IEEE Trans. on Computers*, C-38(12):1612–1630, December 1989.

[15] High Performance Computational Chemistry Group. NWChem: A computational chemistry package for parallel computers, version 1.1, Pacific Northwest Laboratory, Richland, WA 99352, 1995.

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

[17] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proc. 5th ACM SIGPLAN Symp. Principles & Practice of Parallel Programming*, Santa Barbara, CA, pages 179–188, July 1995.

[18] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Languages and Compilers for Parallel Computing,* U. Banerjee et al. (Eds.), Lecture Notes in Computer Science, pages 344–358, 1992.

[19] M. Kandemir, A. Choudhary, J. Ramanujam, and M. Kandaswamy. Locality optimization algorithms for compilation of out-of-core codes. *Journal of Information Science and Engineering*, 14(1):107–138, March 1998.

[20] M. Kandemir, A. Choudhary, J. Ramanujam, and R. Bordawekar. Compilation techniques for out-of-core parallel computations. *Parallel Computing,* 24(3-4):597–628, June 1998.

[21] 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.

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

[23] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proc. Supercomputing'95*, San Diego, CA, December 1995.

[24] J. Laudon and D. Lenoski. The SGI Origin: A cc-NUMA highly scalable server. In *Proc. 24th Annual International Symposium on Computer Architecture*, May 1997.

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

[26] W. Li. *Compiling for NUMA parallel machines*. Ph.D. Thesis, Cornell University, Ithaca, NY, 1993.

[27] J. Li and M. Chen. Compiling communication efficient programs for massively parallel machines. *Journal of Parallel and Distributed Computing*, 2(3):361–376, 1991.

[28] M. Mace. *Memory Storage Patterns in Parallel Processing*, Kluwer Academic Publishers, Boston, MA, 1987.

[29] V. Maslov. Delinearization: an efficient way to break multi-loop dependence equations. In *Proc. SIGPLAN Conf. Programming Language Design & Implementation*, pages 152–161, San Francisco, CA, June 1992.

[30] K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems,* 18(4):424–453, July 1996.

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

[32] D. Palermo and P. Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Proc. 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, pages 392–406, 1995.

[33] J. Ramanujam. *Compile-time techniques for parallel execution of loops on distributed memory multiprocessors.* Ph.D. Thesis, The Ohio State University, Columbus, Ohio, 1990; also available from University Microfilms Inc., as Document 91-11789.

[34] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing 92*, Minneapolis, MN, pages 214–223, November 1992.

[35] J. Ramanujam and A. Narayan. Integrating data distribution and loop transformations for distributed memory machines. In *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing,* D. Bailey *et al.,* Eds., SIAM Press, pages 668–673, February 1995.

[36] 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):472–482, October 1991.

[37] A. Schrijver. *Theory of Linear and Integer Programming*, John Wiley, 1986.

[38] S. Tandri and T. Abdelrahman. Automatic partitioning of data and computations on scalable shared memory multiprocessors. In *Proc. 1997 International Conference on Parallel Processing*, Bloomingdale, IL, pages 64–73, August 1997.

[39] J. Torrellas, M. Lam, and J. Hennessey. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[40] E. Torrie, C. Tseng, M. Martonosi, and M. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proc. International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.

[41] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for for improving data locality on cc-NUMA compute servers. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, pages 279–289, October 1996.

[42] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, Dec 1994.

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

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

Author Biographies

Mahmut Kandemir is a PhD candidate in electrical engineering and computer science department at Syracuse University. He received a BS and an MS, both in computer engineering, from Istanbul Technical University. His research interests include different aspects of locality improvement techniques for cache memories, optimizations for I/O-intensive applications, and computer architecture. He is a student member of the IEEE Computer Society.

Alok Choudhary received his Ph.D. from University of Illinois, Urbana-Champaign, in Electrical and Computer Engineering, in 1989, M.S. from University of Massachusetts, Amherst, in 1986 and B.E.(Hons.) from Birla Institute of Technology and Science, Pilani, India in 1982. He has been an associate professor in the Electrical and Computer Engineering Department at Northwestern University since September, 1996. From 1993 to 1996 he was an associate professor in the ECE department at Syracuse University and from 1989 to 1993 he was an assistant professor in the same department. He has worked in industry for computer consultants prior to 1984.

Alok Choudhary received the National Science Foundation's Young Investigator Award in 1993 (1993-1999). He has also received an IEEE Engineering Foundation award, an IBM Faculty Development award and an Intel Research Council award.

His main research interests are in high-performance computing and communication systems and their applications in many domains including multimedia systems, information processing and scientific computing. In particular, his interests lie in the design and evaluation of architectures and software systems (from system software such as run-time systems, compilers and programming languages to applications), high-performance servers, high-performance databases and input-output. He has published more than 100 papers in various journals and conferences in the above areas. He has also written a book and several book chapters on the above topics. His research has been sponsored by (past and present) DARPA, NSF, NASA, AFOSR, ONR, DOE, Intel, IBM, and TI.

Nagaraj Shenoy

Prithviraj Banerjee received his B.Tech. degree in Electronics and Electrical Engineering from the Indian Institute of Technology, Kharagpur, India, in August 1981, and the M.S. and Ph.D degrees in Electrical Engineering from the University of Illinois at Urbana-Champaign in December 1982 and December 1984 respectively.

Dr. Banerjee is currently the Walter P. Murphy Professor and Chairman of the Department of Electrical and Comp uter Engineering, and Director of the Center for Parallel and Distributed Computing. at Northwestern University in Evanston, Illinois. Prior to that he was the Director of the Computational Science and Engineering program, and Professor of Electrical and Computer Engineering and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign.

Dr. Banerjee's research interests are in Parallel Algorithms for VLSI Design Automation, Distributed Memory Parallel Compilers, and Compilers for Adaptive Computing, and is the author of over 200 papers in these areas. He leads the PARADIGM compiler project for compiling programs for distributed memory multicomputers, the Prop-erCAD project for portable parallel VLSI CAD applications, and the MATCH project on a MATLAB compilation environment for adaptive computing. He is also the author of a book entitled "Parallel Algorithms for VLSI CAD" published by Prentice Hall, Inc., 1994. He has supervised 25 Ph.D. and 30 M.S. student theses so far.

Dr. Banerjee has received numerous awards and honors during his career. He is the recipient of the 1996 Frederick Emmons Terman Award of ASEE's Electrical Engineering Division sponsored by Hewlett-Packard. He was elected to the Fellow grade of IEEE in 1995. He received the University Scholar award from the University of Illinois for in 1993, the Senior Xerox Research Award in 1992, the IEEE Senior Membership in 1990, the National Science Foundation's Presidential Young Investigators' Award in 1987, the IBM Young Faculty Development Award in 1986, and the President of India Gold Medal from the Indian Institute of Technology, Kharagpur, in 1981.

Dr. Banerjee has served as the Program Chair of the Int. Conf. on Parallel Processing for 1995. He has served on the Program and Organizing Committees of the 1988, 1989, 1993 and 1996 Fault Tolerant Computing Symposia, the 1992, 1994, 1995, 1996 and 1997 International Parallel Processing Symposium, the 1991, 1992, 1994 and 1998 International Symposia on Computer Architecture, the 1998 International Conference on Architectural Support of Programming Languages and Operating Systems, the 1990, 1993, 1994, 1995, 1996, 1997 and 1998 International Symposium on VLSI Design, the 1994, 1995, 1996, 1997 and 1998 International Conference on Parallel Process ing, and the 1995, 1996 and 1997 International Conference on High-Performance Computing. He has served as General Chairman of the International Conference on Parallel and Distributed Computing Systems in 1997, and the

International Workshop on Hardware Fault Tolerance in Multiprocessors, 1989. He is an Associate Editor of the Journal of Parallel and Distributed Computing, and IEEE Transactions on Computers. In the past he has served as Associate Editor of the IEEE Transactions on VLSI Systems, and the Journal of Circuits, Systems and Computers. He has been a consultant to many companies and is on the Technical Advisory Board of Ambit Design Systems.

J. Ramanujam received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Madras, in August 1983, and the M.S. and Ph.D. degrees in Computer Science from The Ohio State University, Columbus, Ohio, in August 1987 and December 1990, respectively.

He is currently an an Associate Professor of Electrical and Computer Engineering at Louisiana State University, Baton Rouge. His research interests are in compilers for high-performance computer systems, program transformations, high-level synthesis, parallel input/output systems, parallel architectures and algorithms.

Dr. Ramanujam received the National Science Foundation's Young Investigator Award in 1994. He has served on the Program Committees of the 1997 International International Conference on Parallel Processing and the 8th International Conference on Supercomputing, 1995 and several other conferences. He is a member of the High Performance Fortran Forum. He has taught tutorials on compilers for high-performance computers at several conferences such as the International Conference on Parallel Processing (1998, 1996), Supercomputing 94, Scalable High-Performance Computing Conference (SHPCC 94), and the International Symposium on Computer Architecture (1993 and 1994).

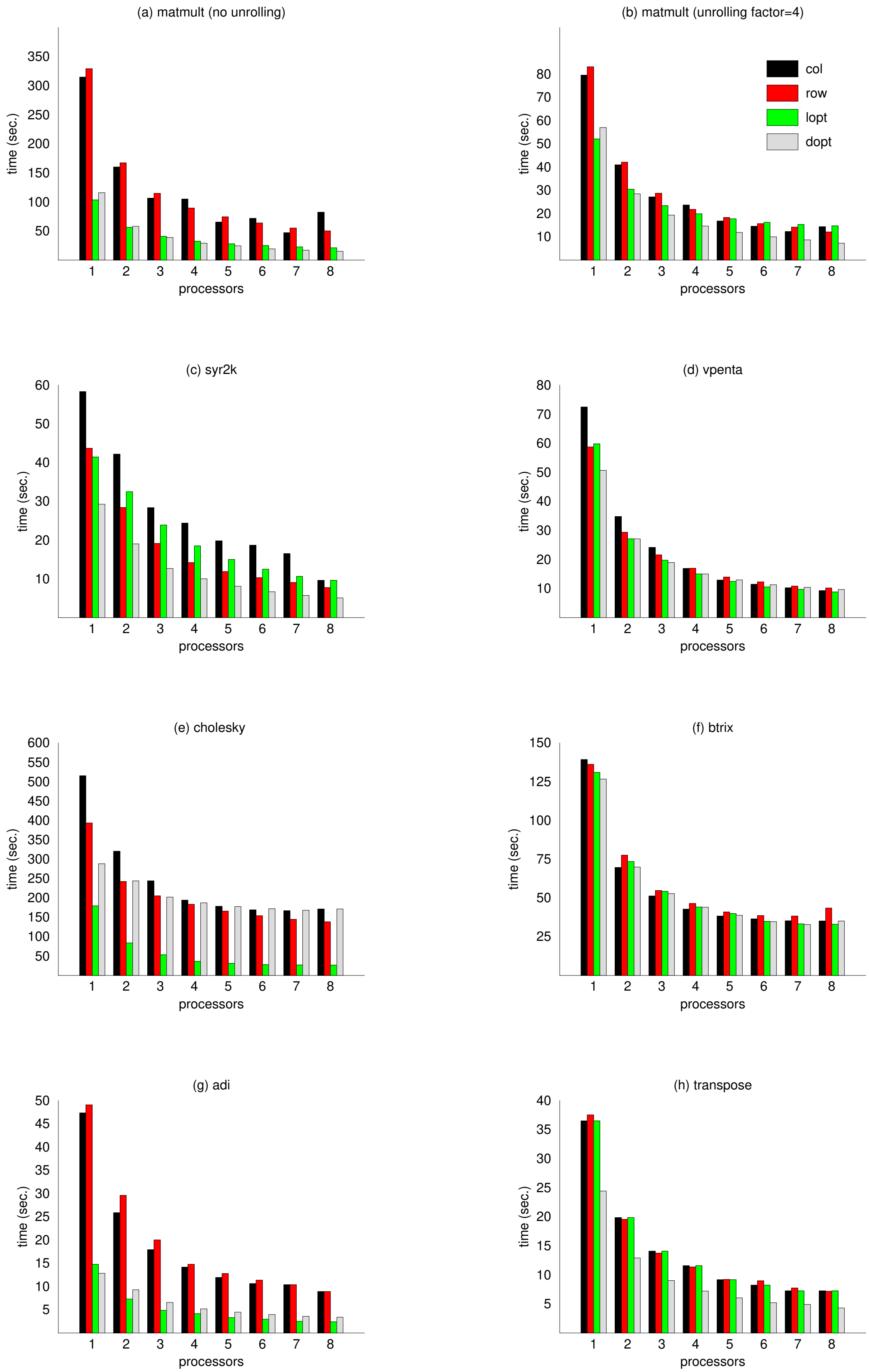

Figure 3: Execution times in Convex Exemplar.

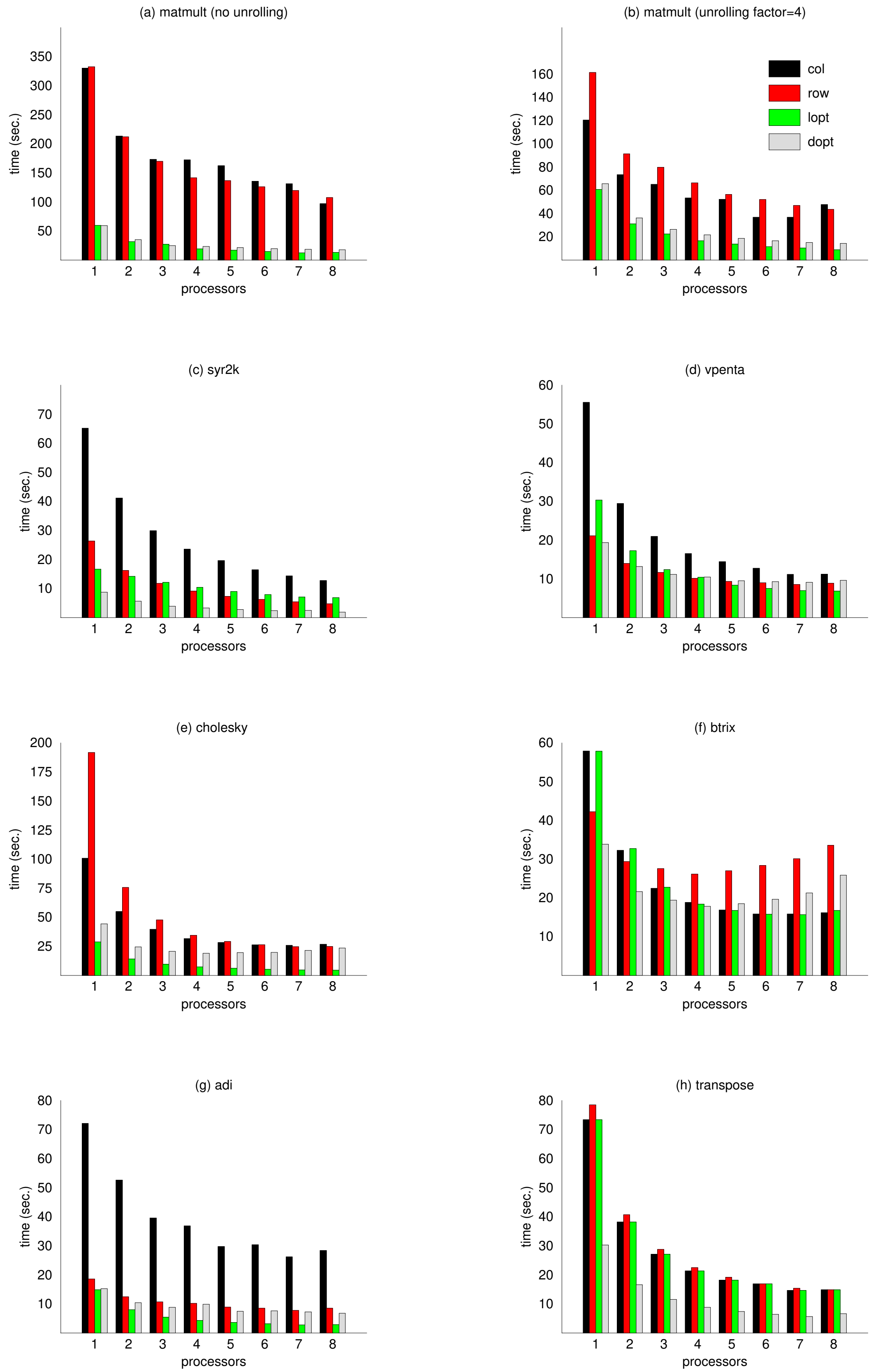

Figure 4: Execution times in SGI Origin 2000.

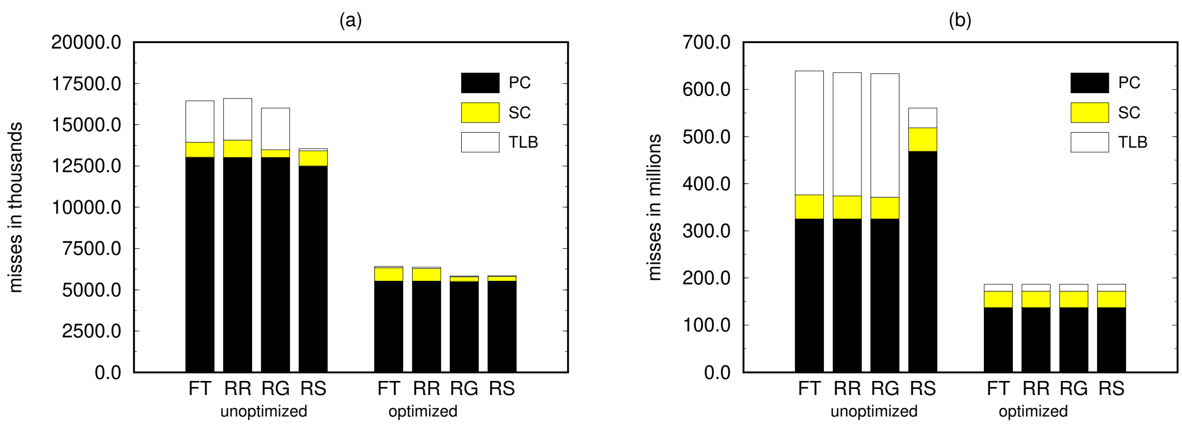

Figure 5: Miss decomposition for different versions of matrix transpose for (a) 1000×1000 and (b) 5000×5000 double-precision matrices on 8 processors of the SGI Origin.

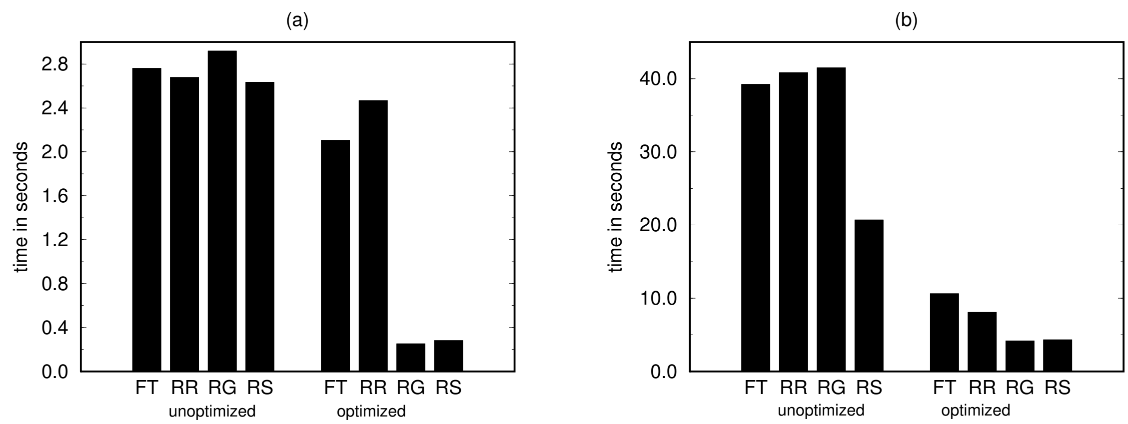

Figure 6: Execution times for different versions of matrix transpose for (a) 1000×1000 and (b) 5000×5000 double-precision matrices on 8 processors of the SGI Origin.

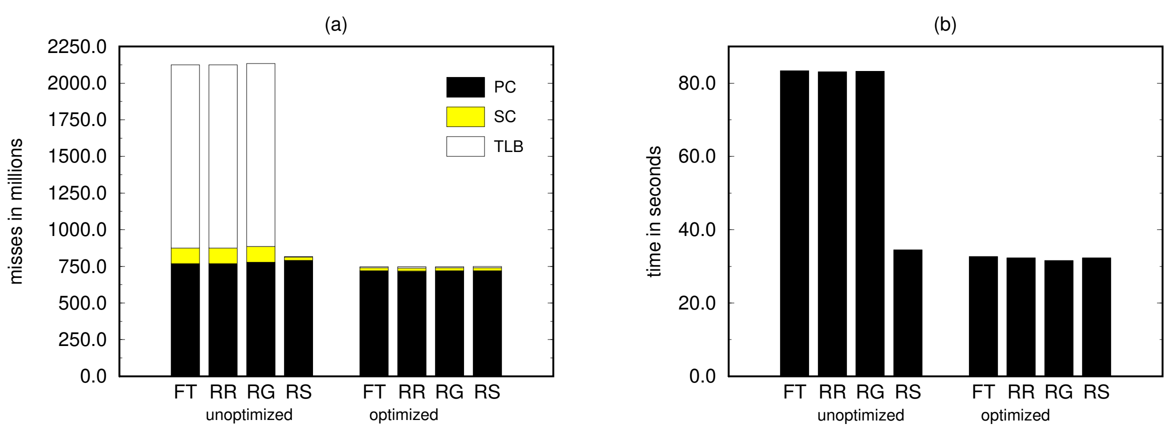

Figure 7: Miss decomposition (a) and execution times (b) for different versions of multiply with 2500×2500 double-precision matrices on 8 processors of the SGI Origin.