A Data Layout Optimization Technique Based on Hyperplanes

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

CPDC-TR-97-04

December 1997

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

A Data Layout Optimization Technique Based on Hyperplanes

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

Abstract

This paper presents a data layout optimization technique based on hyperplane theory from linear algebra. Given a program, our framework automatically determines the optimal 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 layouts, and (2) determining data transformation matrices to implement 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 also give sketch of an algorithm which considers optimizing parallelism and spatial locality simultaneously. Our results on eight programs on the Convex Exemplar and SGI Origin 2000 show that the layout optimizations can be effective in optimizing spatial locality.

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

1 Introduction

In any computer system that contains some sort of memory hierarchy, the accesses to a nearby memory location are always faster than accesses to a farther location. This encourages programmers and compiler writers to attempt to modify the access patterns of a program so that the majority of accesses are made to the nearby memory. Previous research in compilers generally concentrated on iteration space transformations and scheduling techniques to improve locality. Among the techniques used are unimodular [4] and non-unimodular [23, 17] iteration space transformations, tiling [29], loop fusion [20] and affinity scheduling [1]. The common characteristic of these techniques is that they 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 transform data layouts so that better locality is obtained. There are several observations that motivated us to conduct such a study.

- Some programs are not amenable to loop transformations. The data dependences [30] 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 a given loop nest.

- 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 are in general more difficult to optimize using loop transformations whereas in many cases data transformations can successfully be applied to the arrays referenced in them.

- Finally, these two transformation techniques can be combined in a unified way. In fact, it has already been reported [5, 13, 14] that some programs are best improved by a unified transformation approach.

Setting off with these observations, we present a framework which uses data transformations to optimize locality. Our framework is fairly general in the sense that it works on a large search space and considers all sorts of memory layouts that can be expressed by hyperplanes. Specifically, in this paper we make the following contributions:

- We explain how the hyperplane theory can be used for optimizing cache locality.
- We present a method that determines the optimal layouts for all arrays referenced in a given a loop nest.
- We show that for a single loop nest, under specific conditions, data layout transformations can optimize the nest for perfect locality.
- We show how our method can optimize multiple loop nests simultaneously and how it can take into account parallelism as well.
- We present experimental results on two distributed-shared-memory machines, namely Convex Exemplar and SGI Origin to validate our theoretical findings.

In this paper, when we talk about loop transformations we mean linear loop transformations that can be expressed by square non-singular transformation matrices. Similarly, the data transformations we consider can be expressed as linear non-singular square transformation matrices.

The remainder of this paper is organized as follows. In the next section, we outline the notation used and review concepts such as reuse and locality. In Section 3, we present an overview of hyperplane theory. In Section 4, we show how to use that theory to optimize memory layouts of arrays referenced in loop nests. In Section 5, we explain how to obtain a suitable data transformation matrix to generate optimized code. In Section 6, we present experimental results on Convex Exemplar and SGI Origin. In Section 7, we discuss the related work and conclude the paper with a summary in Section 8.

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 $I = (i_1, i_2, ..., i_n)^T$; here, each i_k denotes a loop index with i_1 as the outermost loop and i_n the innermost. In this paper, where there is no confusion, 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. We also assume that all loop bounds and subscript expressions are affine functions of enclosing loop indices. 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 determined by 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 [29, 17]. 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 $X(i - 2j, j + 3)$ appearing in a two-deep loop nest (i, j), we have

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

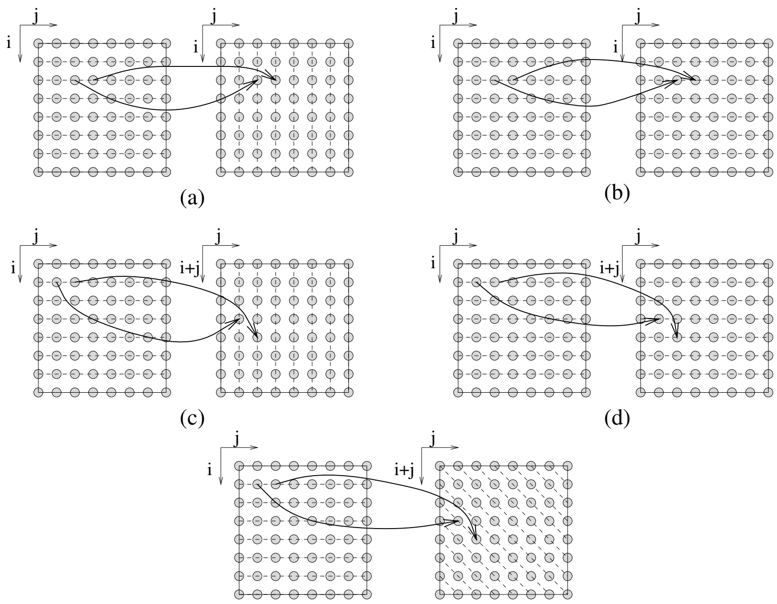

Figure 1: Different access patterns on two-dimensional iteration and data spaces. Two consecutive iterations access (a) two different columns; (b) the same row; (c) the elements in a diagonal (different columns); (d) the elements in a diagonal (different rows); (e) two consecutive elements in the same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

In general, for a reference to an m-dimensional array inside an n-dimensional loop nest, the access matrix is $m \times n$ and the offset vector is $m \times 1$. An important class of references is the class of *uniformly generated references (UGR)*, first defined by Gannon et al. [8].

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 machine which 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* [29]. We have to stress that a program may reuse the 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 7. For this example, interchanging [31] 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 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 $(3, 1)$ 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. In fact, 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 [10, 24]. 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

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

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 $(0, 1, 0)$.

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

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

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 $(3, 8)$ 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 do not bother about the relative orders of hyperplanes, a single hyperplane family is sufficient to define a memory layout partially. In three- or higher-dimensional cases, we may have two 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 whenever two iteration points which have proximity in time access data points (both using R) which 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 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.

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 `h`-dimensional array can be stored in memory in `h!` 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[1]. 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 the optimal memory layouts that are defined by hyperplanes; and

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

Each sub-problem can be solved independently. Previously, [5], [13], [14] and [16] offered algorithms to handle the first sub-problem whereas [22] and [16] 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.

Since a one-dimensional array can only be stored in one way, the data space transformations are only meaningful for two or higher dimensional arrays[2].

Our main objective in this paper is to solve the first subproblem mentioned above, namely, finding the 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 within a compilation framework. We choose to separate the problem of determining the 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.

We define our problem 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.

4.2 Determining the Optimal Layouts

In this subsection, we present our data space restructuring framework in detail. In particular, we show how to determine the 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 $\bar{I}_{next}$

[1] 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.

[2] The data layout transformations can also be applied to one-dimensional arrays as well provided that they can be delinearized first [6]. We are currently working on a 'cache conscious' delinearization scheme based on Maslov's algorithm [19].

of a given loop nest of depth n. Such two iterations have identical values for each loop index except for the innermost loop, i.e.,

$$\bar{I} = \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_{n-1} \\ i_n \end{pmatrix} \text{ and } \bar{I}_{next} = \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_{n-1} \\ 1 + i_n \end{pmatrix}$$

In order to exploit the locality for a reference R denoted by access matrix A_R, two consecutive iterations $\bar{I}$ and $\bar{I}_{next}$ 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 $\bar{I}_{next} - \bar{I} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$, we have $g^T(A_R \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}) = 0 \Longrightarrow g^T\bar{a_n} = 0 \Longrightarrow a_n\bar{g} = 0 \Longrightarrow \bar{g} \in Ker\{a_n\}$

□

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, $\bar{g_U} \in Ker\{(0,1)\} \Longrightarrow \bar{g_U}^T = (1,0)$; $\bar{g_V} \in Ker\{(1,0)\} \Longrightarrow \bar{g_V}^T = (0,1)$; $\bar{g_W} \in Ker\{(1,0)\} \Longrightarrow \bar{g_W}^T = (0,1)$; $\bar{g_X} \in Ker\{(1,1)\} \Longrightarrow \bar{g_X}^T = (1,-1)$; and $\bar{g_Y} \in$

$Ker\{(-1,1)\} \Longrightarrow \bar{g_Y}^T = (1,1)$. 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 [13] and [5] as neither of those approaches can detect skewed (diagonal) layouts. 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),

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

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

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)	$V(k,i-1,j+k)$
$(2,1,1)$	$V(1,1,2)$	$(2,2,1)$	$V(1,1,3)$	$(3,1,1)$	$V(1,2,2)$
$(2,1,2)$	$V(2,1,3)$	$(2,2,2)$	$V(2,1,4)$	$(3,1,2)$	$V(2,2,3)$
$(2,1,3)$	$V(3,1,4)$	$(2,2,3)$	$V(3,1,5)$	$(3,1,3)$	$V(3,2,4)$
$(2,1,4)$	$V(4,1,5)$	$(2,2,4)$	$V(4,1,6)$	$(3,1,4)$	$V(4,2,5)$
...	...	...	...	...	...

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.

Lemma 2 *A given 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

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

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:

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

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

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

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

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

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 [30, 31] 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 as well.

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 specific conditions, can optimize an array which has multiple (different–not necessarily uniformly generated) access matrices in the program. These specific conditions are discussed in Section 4.7.

4.5 Temporal Locality

If a reference has temporal locality in the innermost loop, then an optimizing compiler can keep it in a register during the entire execution of the innermost loop. In fact, 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. 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 as well. 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, compiler selects 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 [16], 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 solution process. As an example, consider the following program fragment.

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

do k = lk, uk
  do l = ll, ul
    ...  U(l,l) ...
  enddo
enddo
```

In this example, the array U is referenced in both these nests. The original access matrices are

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

The compiler handles this program as if there is an extra outermost loop with trip count zero enclosing both the nests. In that case, the modified access matrices are

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

where the first zero columns correspond to that extra loop. As far as our method is concerned, nothing is changed; and as before, this array can be stored in memory diagonally as the last columns of access matrices of the references are the same.

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. Since to optimize spatial locality for the innermost loop, our approach focuses on the last columns of access matrices, intuitively, if the Ker sets of the last columns of two references are equal, there will be no conflict in optimizing both. This can be stated as follows for a general case.

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 row representations of the last columns of these access matrices respectively. Then, these references can be optimized in a conflict-free manner if*

$$Ker\{a_{l_1}\} = Ker\{a_{l_2}\} = ... = Ker\{a_{l_s}\}$$

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 UGRs have the same last columns in their access matrices. $\square$

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

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

Since, $Ker\{a_{l_1}\} = Ker\{a_{l_2}\} = Ker\{a_{l_3}\} \neq Ker\{a_{l_4}\}$, 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 and decides diagonal layout with the hyperplane $(1, -1)$ instead of $(2, -1)$. Since, in practice, it is rare that an array is referenced with so many different access matrices, this approach seems reasonable. In general case, however, whenever the conflicting references occur in different nesting levels, a *prioritizing* scheme should be adopted. Investigating such a scheme however is beyond the scope of this paper. Our optimization strategy can easily be modified to take a suitable prioritizing scheme into account. Ideally, the references should be prioritized according to their frequency of access. But, in practice, unknown loop bounds, complex subscript expressions and conditionals make it difficult to estimate at compile-time the number of times a reference will be executed. Our current technique orders references by nesting level; that is, a reference located within a deeper loop has priority over the one with a smaller nesting level. To break a tie, we also take into account the number of times the references occur, the bounds of the surrounding loops (if known) and the sizes of the associated arrays in that order. Although, this ordering cannot be optimal for every case, we found that in practice it is highly successful.

Contrast the last example with the following which also contains a non-unit strided 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, $Ker\{a_{l_1}\} = Ker\{a_{l_2}\} = Ker\{a_{l_3}\} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, 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 can be extended to handle non-unit stride accesses as well by shrinking arrays whenever possible. In the next subsection, we focus on this issue.

4.8 Unit-Stride Accesses

In the last example, for the reference $U(i, 2j)$, our approach has found row-major layout suitable for exploiting spatial locality. In fact, for any reference $U(i, vj)$ where v is a constant, the approach offers the row-major layout as the best alternative. While this is reasonable in general, in cases where v is larger than the number of elements residing in a cache line, no spatial locality will be exploited. If v is smaller than size of the cache line, however, some spatial locality will be exploited, but still the cache line will be underutilized (unless $v = 1$). Suppose there are s references of the form $U(i, v_k j)$ where $1 \leq k \leq s$, we can choose to *shrink* the array by $v_g = \gcd(v_1, v_2, ..., v_s)$, assuming as if there is a single reference $U(i, v_g s)$. We now illustrate how our approach handles array shrinking automatically.

Let us now return to the last example and concentrate on $U(i, 2j)$ again assuming that it is the only reference inside the nest. A more detailed explanation on how the row-major layout has been determined for this reference is as follows: Consider two consecutive iterations of the loop nest: (i, j) and $(i, j+1)$. For spatial locality, the following condition should hold:

$$(g_1, g_2) \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = (g_1, g_2) \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} i \\ j+1 \end{pmatrix}$$

which in turn means that

$$(g_1, g_2) \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \Longrightarrow (g_1, g_2) \begin{pmatrix} 0 \\ 2 \end{pmatrix} = 0 \Longrightarrow g_2 = 0 \Longrightarrow g_1 = 1.$$

What this implies that the two data elements will have locality if they are mapped onto the same row. In general, the hyperplanes in the data space correspond to rows of the array, columns of the array, etc. Therefore, for our example, the following expression should yield an integer for all possible data elements (d_1, d_2):[3]

$$(g_1, g_2) \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}.$$

Since, in our example, $(g_1, g_2) = (1, 0)$, the expression above results in d_1 which is an integer.

We note that although two data elements might have spatial locality, that does not necessarily mean that they are consecutive. In particular, as in this example, there may be a gap between two data elements which have locality. Depending on the size of this gap (which is determined by the stride in the access), the cache locality may or may not be good.

Now we can think of another set of hyperplanes (in this case, orthogonal to the layout hyperplanes – that is, column hyperplanes) such that two data elements in a row will be consecutive in this new hyperplane set. In our example, this condition can be expressed as follows:

$$(z_1, z_2) \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + 1 = (z_1, z_2) \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} i \\ j+1 \end{pmatrix}$$

which in turn means that

$$(z_1, z_2) \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 1 \Longrightarrow (z_1, z_2) \begin{pmatrix} 0 \\ 2 \end{pmatrix} = 1 \Longrightarrow 2z_2 = 1 \Longrightarrow z_2 = 1/2.$$

Setting z_1 to 0, this new set of hyperplanes can be expressed by $(0, 1/2)$. As in the previous case, the following should yield an integer:

$$(z_1, z_2) \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}.$$

[3] Notice that the hyperplane coefficients themselves do not need to be integers as long as the resulting values are integers.

Therefore, we conclude that $d_2/2$ is an integer; that is, $d_2 = 2k$ for an integer k. What this says is that for the data elements that should be stored contiguously (in a row), the second indices must differ by two. Thus, the array should be shrunk by a factor of two in the second dimension. A method of incorporating this information into the compilation framework to generate appropriate code will be explained in the next section.

Lemma 4 *Let the array to be shrunk have an* integer $m \times n$ *access matrix* $A = [a_{ij}]$. *Then we can shrink this array by a hyperplane* $(0, 0, \cdots, 0, k)$ *where* $0 < k < 1$ *if* $a_{mn} > 1$ *and* $a_{mj} = c_j a_{mn}$ *holds for each* $1 \le j \le n$ *for some integer* c_j.[4]

Proof: *(Sketch)* Let $\bar{z}^T = (0, 0, \cdots, 0, k)$ where $k \neq 0$. First we show that such a $\bar{z}^T$ can always be found and then we prove that $\bar{z}^T A_U (0, 0, \cdots, 0, 1)^T = 1$ under the given condition on A_U.

From the preceding discussion, we know that $\bar{z}^T \bar{a} = 1$ where $\bar{a} = (a_{1n}, a_{2n}, \cdots, a_{mn})^T$ is the last column of the access matrix. If $\bar{z}^T = (\gamma_1, \gamma_2, \cdots, \gamma_m)$, then

$$a_{1n}\gamma_1 + a_{2n}\gamma_2 + \cdots + a_{mn}\gamma_{mn} = 1.$$

In this equation, if a_{mn} is 0 then the last column has temporal locality in the innermost loop; we just omit this column. Otherwise, we choose $\gamma_1 = 0$, $\gamma_2 = 0$, $\cdots$,$\gamma_{m-1} = 0$, and $\gamma_m = 1/a_{mn}$ which means $\bar{z}^T = (0, 0, \cdots, 0, 1/a_{mn})$. We write $\bar{z}^T A_U (0, 0, \cdots, 0, 1)^T$ in open form as follows:

$$(0, 0, \cdots, 0, 1/a_{mn}) \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2(n-1)} & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ k_1 a_{mn} & k_2 a_{mn} & \cdots & k_{n-1} a_{mn} & a_{mn} \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}.$$

It is easy to see that this expression evaluates to 1. Therefore, under the given condition on the access matrix, it is possible to choose a hyperplane to shrink the array in question. □

For the remainder of the paper we call the vector $(0, 0, \cdots, 0, 1/a_m n)$ *shrinking vector*.
Consider now the following loop nest.

```
do i = li, ui
  do j = lj, uj
    do k = lk, uk
      ··· U(3i,4j-2k,5k+5i-10j) ···
    enddo
  enddo
enddo
```

The access matrix is

$$A_U = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 4 & -2 \\ 5 & -10 & 5 \end{pmatrix}.$$

Since both a_{31} and a_{32} are multiples of a_{33}, the compiler can shrink this array using shrinking vector $(0, 0, 1/5)$.

[4] Notice that deciding whether or not an array should be shrunk is a different topic and beyond the scope of this paper. Here we only show, when decided, how we can shrink the array.

4.9 Multiprocessor case

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 [27].
- 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 (see for example Li's thesis [17]) parallelizing a loop which 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 [26]. Since our framework causes spatial locality to be carried by the innermost loops, in most cases, the compiler can safely parallelize (provided that the dependences allow) the outermost loops without an apparent danger of false sharing.

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

We now present sketch of an algorithm which can optimize spatial locality and at the same time reduce the false sharing for the cases where outermost loop parallelism cannot be obtained. 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 [29]. From this condition, since A_{U_s} represents a linear mapping,

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

where $r_{\bar{U}s} = \bar{J} - \bar{I}$.

In that case, $r_{\bar{U}s}$ is termed as the spatial reuse vector and $Ker\{A_{U_s}\}$ is referred as spatial reuse vector space [29]. When there is no confusion, we use $r_{\bar{U}}$ instead of $r_{\bar{U}s}$ for the sake of clarity. The loop which corresponds to the first non-zero element of the generator vector of $Ker\{A_{U_s}\}$ is said to carry spatial reuse.

Here, 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 5 *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. 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

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

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

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 any vector from $Ker\{A_U \bar{r_U}'\}$ is the desired hyperplane vector which 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.

In a shared-memory multiprocessor case, we should take into account the issues related to parallelism as well. As a rule, to prevent a common form of false sharing, a loop which carries spatial reuse should not be parallelized [17]. 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 [7, 11].

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.

Our 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 a conflict resolution scheme should be applied. 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 and a set of basis vectors for it. These basis vectors represent the layout of the array.

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

Figure 2: Algorithm for optimizing spatial locality.

It should be noted that we do not attempt to try all possible spatial reuse vectors. Instead we try only n of them. Since the optimizing compilers for shared-memory multiprocessors are quite successful in parallelizing the outermost loops, the algorithm terminates quickly; and the spatial locality in the innermost loops is exploited without incurring severe false sharing.

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

5 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 [22] 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, $L_{U_{default}}$ is the default layout (which we assume for now, without loss of generality, column-major following the Fortran convention); and $L_{U_{desired}}$ 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

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

When this equation is solved for the elements of M_U, some of the elements of M_U (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.

We consider the following program once more, assuming that the default layout is column-major; that is $L_{U_{default}} = (0, 1)$.

```
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

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

Using equation (5),

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

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

$$(0,1)M_W = (0,1) \implies M_W = \begin{pmatrix} \times & \times \\ 0 & 1 \end{pmatrix}$$

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

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

where $\times$ stands for an unknown entry. These matrices should be completed such that they should be non-singular. Fortunately, the completion algorithms offered by [17] 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:

- $M_U A_U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
- $M_V A_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
- $M_W A_W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
- $M_X A_X = \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
- $M_Y A_Y = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$

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), 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 as well.

In order to illustrate the generality of the approach, we next consider the following loop nest under the assumption that the default layout is column-major.

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

In this example, $L_{U_{default}} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ and $L_{U_{desired}} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$. Therefore, from $L_{U_{default}} M_U = L_{U_{desired}}$, $M_U = \begin{pmatrix} \times & \times & \times \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$. An example solution is $M_U = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

The transformed access matrix is

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

And similarly, $L_{V_{default}} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ and $L_{V_{desired}} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix}$.

From $L_{V_{default}} M_V = L_{V_{desired}}$, $M_V = \begin{pmatrix} \times & \times & \times \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}$. An example solution is $M_V = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}$.

The transformed access matrix is

$$M_V A_V = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

The transformed program is as follows.

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

We now consider the problem of code generation when an array is shrunk. The layout constraint matrix of the transformed array (when shrunk) is the same as the one that is found when shrinking is not taken into account except that the coefficients of the shrinking vector (hyperplane which is used for shrinking) are added as the last row to the layout constraint matrix. Also, assuming that the original layout is column-major, the original layout constraint matrix should be completed to permutation matrix by adding the vector $(1, 0, \cdots, 0, 0)$ as the last row.

Let us consider the following example:

```
do i = 1, n
  do j = 1, n/2
    ...  U(3i+2j,5j) ...
  enddo
enddo
```

Without taking stride into account, the straightforward application of our method gives $(5, -2)$ as $L_{U_{desired}}$. Since $L_{U_{default}}$ is $(0, 1)$, the data transformation matrix is

$$M_U = \begin{pmatrix} 1 & 0 \\ 5 & -2 \end{pmatrix}.$$

The transformed reference is $U(3i + 2j, 15i)$. We note that the stride for the innermost loop in the transformed access is 2.

Now we demonstrate the effect of shrinking. We note that this array can be shrunk using the hyperplane $(0, 1/5)$. In that case,

$$L_{U_{desired}} = \begin{pmatrix} 5 & -2 \\ 0 & 1/5 \end{pmatrix} \text{ and } L_{U_{default}} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

The data transformation matrix, on the other hand, is

$$M_U = \begin{pmatrix} 0 & 1/5 \\ 5 & -2 \end{pmatrix}.$$

In that case, the transformed reference is $U(j, 15i)$. We note that the access is with unit-stride. Similarly, a reference such as $U(3i, 4j - 2k, 5i + 5k - 10j)$ in a three-deep loop nest (i, j, k) is transformed to $U(4j - 2k, 6i + 16j, 3i)$ with straightforward approach; and to $U(i - 2j + k, 10i, 3i)$ by taking stride into account. Notice that in the latter case, the spatial locality is exploited in the innermost loop with the unit-stride.

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 [3, 30] in dependence testing. Given an affine function of a number of variables and inequalities which 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 [3, 30]). In this method, the computed extreme values may not be tight as could have been if the Fourier-Motzkin elimination [25] had been used. However, for many programs, applying Fourier-Motzkin elimination to obtain tight bounds may be to 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 $\{1 \leq x \leq 10 \text{ and } x + 1 \leq y \leq 30 - x\}$.

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

$$f_1(i,j,k) = j+k, f_2(i,j,k) = -j, \text{ and } f_3(i,j,k) = i$$

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 follows.

```
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 be quite useful in a compilation environment which considers dynamic layout transformations between loop nest boundaries. In such an environment, 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.

6 Experimental Results

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

6.1 Experimental Suite

We present performance results for eight programs: matmult is a matrix–multiplication routine with two versions (with and without loop unrolling)[5]; syr2k is banded matrix update routine from BLAS; adi is

[5] The unrolled version is called matmult/u.

from Livermore kernels; btrix, vpenta and cholesky are from Spec92/NASA benchmark suite; and transpose is a matrix transpose program from NWChem [21], a software package for computational chemistry. matmult, syr2k and transpose are programs consisting of a single loop nest whereas the others contain multiple loop nests.

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 [17] or allowing the native compiler to derive the best order.

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.

6.2 Platforms

We report execution times in seconds for up to 8 processors in a Convex Exemplar and an SGI Origin, two widely used 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 can have up to 16 processors, 16GB of memory and a CTI cache of size 256MB. The programs are compiled by the native C compiler with -O3 option. The timings are taken using the CXpa performance analyzer.

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

6.3 Performance

Table 3 shows the single processor miss rates obtained using an enhanced version of DineroIII [9] 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 [17]), 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 (jki order for column-major layout from [17]) 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.

Table 3: Uniprocessor miss rates for the programs in our experiment suite [`bs` = block (line) and `cs` = cache size]. [The problem sizes (in doubles) 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	0.441	0.317	0.226	0.223	0.394	0.271	0.156	0.154	0.383	0.259	0.138	0.136	0.012	0.011	0.013	0.011
matmult	32	0.382	0.320	0.182	0.179	0.336	0.274	0.100	0.098	0.324	0.262	0.080	0.078	0.014	0.012	0.014	0.012
matmult	64	0.357	0.327	0.165	0.162	0.312	0.281	0.078	0.076	0.300	0.270	0.056	0.054	0.020	0.019	0.021	0.019
matmult	128	0.354	0.339	0.167	0.164	0.310	0.296	0.078	0.076	0.299	0.285	0.056	0.054	0.033	0.034	0.035	0.033
matmult/u	16	0.255	0.317	0.287	0.177	0.185	0.130	0.101	0.072	0.167	0.084	0.054	0.045	0.013	0.014	0.016	0.012
matmult/u	32	0.245	0.321	0.274	0.158	0.175	0.133	0.088	0.058	0.158	0.087	0.042	0.033	0.016	0.015	0.017	0.013
matmult/u	64	0.219	0.327	0.274	0.155	0.144	0.141	0.071	0.057	0.125	0.094	0.041	0.032	0.022	0.022	0.023	0.019
matmult/u	128	0.217	0.341	0.283	0.166	0.141	0.155	0.099	0.068	0.121	0.109	0.052	0.043	0.035	0.037	0.038	0.034
syr2k	16	0.664	0.417	0.365	0.206	0.658	0.291	0.316	0.179	0.527	0.173	0.281	0.157	0.100	0.041	0.131	0.013
syr2k	32	0.662	0.542	0.187	0.109	0.655	0.353	0.160	0.092	0.514	0.105	0.142	0.080	0.058	0.024	0.067	0.007
syr2k	64	0.660	0.628	0.101	0.066	0.650	0.416	0.082	0.049	0.501	0.091	0.073	0.043	0.044	0.019	0.034	0.004
syr2k	128	0.659	0.646	0.072	0.057	0.640	0.525	0.043	0.030	0.492	0.246	0.038	0.025	0.051	0.023	0.018	0.003
vpenta	16	0.652	0.502	0.502	0.457	0.652	0.498	0.498	0.457	0.435	0.403	0.403	0.311	0.218	0.272	0.272	0.207
vpenta	32	0.543	0.321	0.321	0.251	0.544	0.317	0.317	0.250	0.475	0.294	0.294	0.177	0.133	0.166	0.166	0.131
vpenta	64	0.489	0.236	0.236	0.151	0.489	0.226	0.226	0.148	0.487	0.217	0.217	0.111	0.236	0.113	0.113	0.072
vpenta	128	0.462	0.227	0.227	0.139	0.462	0.181	0.181	0.098	0.487	0.177	0.177	0.078	0.433	0.087	0.087	0.043
cholesky	16	0.615	0.730	0.206	0.352	0.481	0.575	0.183	0.285	0.263	0.326	0.148	0.192	0.167	0.189	0.082	0.133
cholesky	32	0.566	0.706	0.158	0.177	0.484	0.658	0.104	0.143	0.340	0.372	0.074	0.096	0.114	0.168	0.041	0.066
cholesky	64	0.561	0.659	0.169	0.089	0.529	0.658	0.098	0.072	0.362	0.411	0.038	0.048	0.144	0.200	0.021	0.033
cholesky	128	0.411	0.593	0.190	0.045	0.349	0.590	0.096	0.036	0.237	0.445	0.020	0.024	0.099	0.304	0.011	0.017
btrix	16	0.228	0.221	0.223	0.223	0.196	0.180	0.196	0.182	0.137	0.099	0.137	0.120	0.118	0.075	0.118	0.091
btrix	32	0.142	0.152	0.142	0.135	0.117	0.109	0.117	0.100	0.079	0.053	0.079	0.064	0.066	0.039	0.066	0.048
btrix	64	0.108	0.139	0.103	0.099	0.075	0.085	0.075	0.061	0.049	0.040	0.049	0.036	0.039	0.020	0.039	0.026
btrix	128	0.118	0.243	0.110	0.110	0.054	0.096	0.054	0.043	0.033	0.040	0.033	0.022	0.024	0.017	0.024	0.014
adi	16	0.493	0.466	0.350	0.388	0.465	0.349	0.234	0.267	0.310	0.310	0.156	0.178	0.282	0.301	0.156	0.162
adi	32	0.375	0.479	0.319	0.343	0.312	0.330	0.119	0.135	0.202	0.286	0.079	0.090	0.182	0.280	0.078	0.082
adi	64	0.272	0.475	0.322	0.346	0.160	0.367	0.132	0.134	0.102	0.306	0.040	0.047	0.091	0.302	0.039	0.043
adi	128	0.356	0.555	0.412	0.452	0.186	0.465	0.249	0.278	0.053	0.311	0.022	0.025	0.046	0.307	0.020	0.023
transpose	16	0.625	0.625	0.625	0.501	0.375	0.375	0.375	0.250	0.281	0.281	0.281	0.250	0.258	0.258	0.258	0.250
transpose	32	0.813	0.813	0.813	0.753	0.344	0.344	0.344	0.125	0.180	0.180	0.180	0.125	0.139	0.139	0.139	0.125
transpose	64	0.906	0.906	0.906	0.874	0.453	0.453	0.453	0.063	0.130	0.130	0.130	0.063	0.080	0.080	0.080	0.063
transpose	128	0.953	0.953	0.953	0.938	0.445	0.445	0.445	0.031	0.104	0.104	0.104	0.031	0.050	0.050	0.050	0.031

The benchmarks `vpenta`, `cholesky` and `btrix` are relatively large programs consisting several nests. In Convex, `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 with 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 Origin are given in Figure 4. As on the Exemplar, 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.

6.4 Observations

From our experiments, we observe the following:

- The data layout transformations are very effective in optimizing cache locality for uniprocessors. To be precise, five out of eight cases in Convex as well as Origin, the `dopt` version outperforms the other versions (including `lopt`) in a single node.
- Our transformations can optimize some programs such as `syr2k` and `transpose` which cannot be fully optimized by loop transformations.
- In programs where there are multiple 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, in cases where outermost loop parallelism cannot be obtained, the performance may be rather poor. Also, in some cases, the loop transformations may also improve parallelism; thus, may have additional advantage over data transformations (as in `vpenta`).
- The relative performances of different versions across two platforms are different. We attribute this to the different cache sizes, page sizes and memory organizations of the Exemplar and Origin.

7 Related Work

Loop transformations have been used for optimizing cache locality in several papers [17, 29, 20]. Results have shown that on several architectures the speedups achieved by loop transformations alone can be quite large. McKinley *et al.* [20] 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 [29] 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 [17] 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 [29], [17], [20] 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 [22] 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 which 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 [28]. 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 [12] 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 [5] 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; which in turn implies either availability of these bounds during compilation or manipulation of symbolic expressions.

Leung and Zahorjan [16] 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.

Finally, previous work of the authors [13, 14] considers only dimension re-indexing and and like [5] uses a sort of exhaustive search to detect array layouts. In contrast to [13, 14], and [5], the approach explained in this paper considers a much larger search space for memory layouts, and finds the optimal layouts in a single step.

8 Summary

In this paper, we presented an approach based on hyperplane theory and available 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 the optimal layouts for each array and (2) implementing the optimal layouts within a compiler which 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 given a loop nest, data layout transformations can be more effective than iteration space transformations, provided that some conditions are satisfied. Experimental results in Convex Exemplar and SGI Origin 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. In the future, we intend to represent loop transformations for locality with hyperplanes, and integrate loop and data transformations in a unified framework based on hyperplanes.

References

[1] B. Appelbe, and B. Lakshmanan. Optimizing parallel programs using affinity regions. *Technical Report GIT-ICS-92/59, Georgia Institute of Technology*, November 1992.

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

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

[4] U. Banerjee. Unimodular transformations of double loops. In *Proc. Advances in Languages and Compilers for Parallel Processing*, edited by A. Nicolau et.al. MIT Press, 1991.

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

[6] M. Cierniak, and W. Li. Recovering logical data and code structures. Technical Report 591. Dept. of Computer Science, University of Rochester, July 1995.

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

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

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

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

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

[12] Y.-J. Ju, and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Proc. Fourth Intl. Workshop in Languages and Compilers for Parallel Computing*, Ithaca, 1992.

[13] 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.

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

[15] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A data layout optimization technique based on hyperplanes. Technical Report, CPDC-TR-97-04, Northwestern University, December 1997. `ftp://gate.ee.lsu.edu/pub/jxr/papers/tr/cpdc974.ps.`

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

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

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

[19] V. Maslov. Delinearization: an efficient way to break multi-loop dependence equations. In *Proc. the SIGPLAN'92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

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

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

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

[23] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing 92*, pages 214–223, Nov 1992.

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

[25] A. Schrijver. *Theory of Linear and Integer Programming*, Wiley-Interscience series in Discrete Mathematics and Optimization, John Wiley and Sons, 1986.

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

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

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

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

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

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

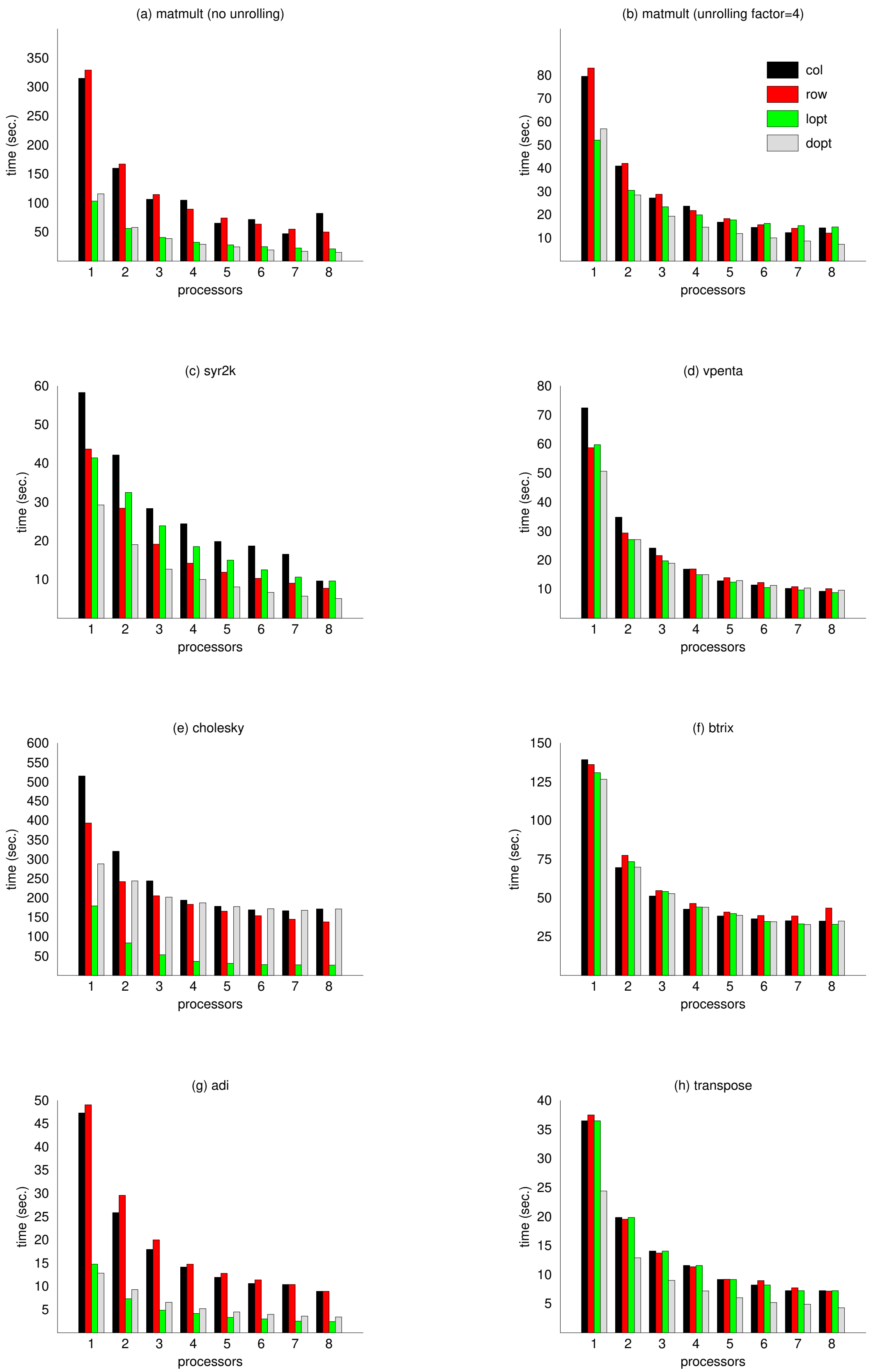

Figure 3: Execution times in Convex Exemplar.

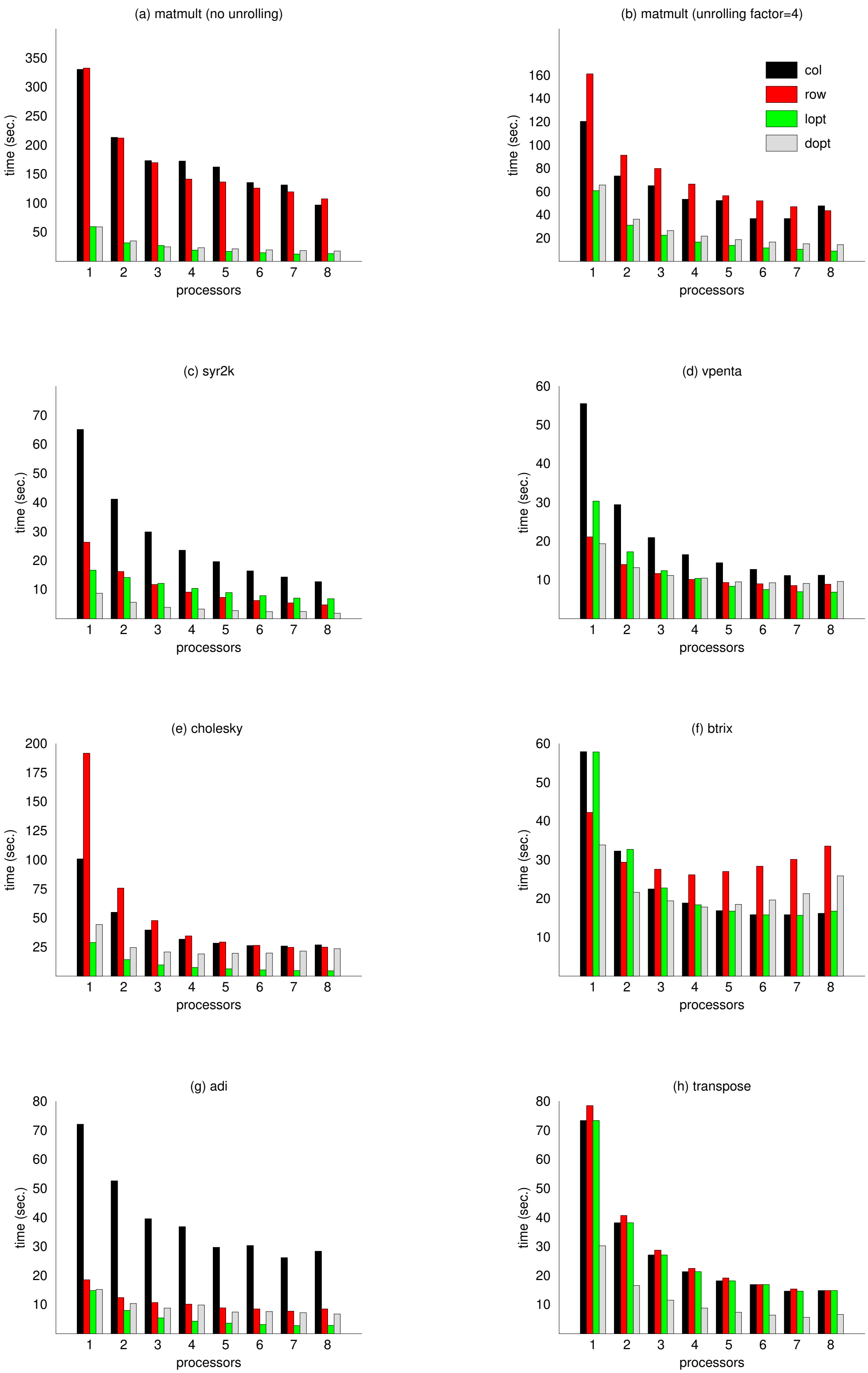

Figure 4: Execution times in SGI Origin.