Proc. 11th ACM international Conference on Supercomputing, Melbourne, Australia, July 1998

A Hyperplane Based Approach for Optimizing Spatial Locality in Loop Nests

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

Abstract

This paper presents a data layout optimization technique based on the theory of hyperplanes 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 specific conditions a loop nest can be optimized for perfect spatial locality by using data transformations. 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. Our results on eight programs on SGI Origin 2000 distributed-shared-memory multiprocessor show that the layout optimizations are effective in optimizing spatial locality.

1 Introduction

On most modern machines, the accesses to a nearby memory location are always faster than accesses to a farther location. This encourages programmers and compiler writers 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 and non-unimodular [12] iteration space transformations, tiling [19], and loop fusion [13]. All these techniques focus on improving data locality *indirectly* as a result of modifying the iteration space traversal order.

In this paper, we take a more direct approach to the data locality optimization problem. Unlike traditional compiler techniques, we focus directly on the data space, and attempt to transform data layouts so that better locality is obtained. There are several observations that motivated this approach. First, some programs are not amenable to loop transformations. The data dependences 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. Secondly, for some programs, even though an iteration space transformation is legal, there may be a data space transformation which results in better locality. In addition, unlike loop transformations, data transformations do not affect all the arrays accessed in a given loop nest. Finally, imperfectly nested loops are in general more difficult to optimize using loop transformations whereas in many cases data transformations can be successfully applied to the arrays referenced in them.

Considering these observations, we present a framework that uses data transformations to optimize locality. Our framework is fairly general in the sense that it works on a large search space and considers various memory layouts that can be expressed by hyperplanes. Specifically, we make the following contributions:

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

We present experimental results on the SGI Origin 2000 to validate our theoretical findings. In this paper, when we talk about loop transformations we mean linear transformations of the iteration space that can be expressed by square non-singular transformation matrices. 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 Section 2 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 the SGI Origin. In Section 7, we briefly discuss the related work and conclude the paper with a summary in Section 8.

2 Technical Preliminaries

We view the iteration space of a loop nest of depth n as an n-dimensional polyhedron where each point is denoted by an $n \times 1$ column vector $\bar{I} = (i_1, i_2, ..., i_n)^T$; here, each i_k denotes a loop index with i_1 as the outermost loop and i_n the innermost. In this paper we will use $(i_1, i_2, ..., i_n)$ to denote a loop nest as well as a point in the iteration space. We show the lower and upper limits for a loop `i` as `li` and `ui` respectively. We assume that the loops are normalized such that the step size is one. We also assume that all loop bounds and subscript expressions are affine functions of enclosing loop indices and symbolic constants. 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

[*]EECS Dept., Syracuse University, Syracuse, NY 13244. e-mail: mtk@ece.nwu.edu

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

[‡]ECE Dept., Northwestern University, Evanston, IL 60208. e-mail: {banerjee,choudhar,nagaraj}@ece.nwu.edu

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

[¶]ECE Dept., Louisiana State University, Baton Rouge, LA 70803. e-mail: jxr@ee.lsu.edu. Supported in part by NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422.

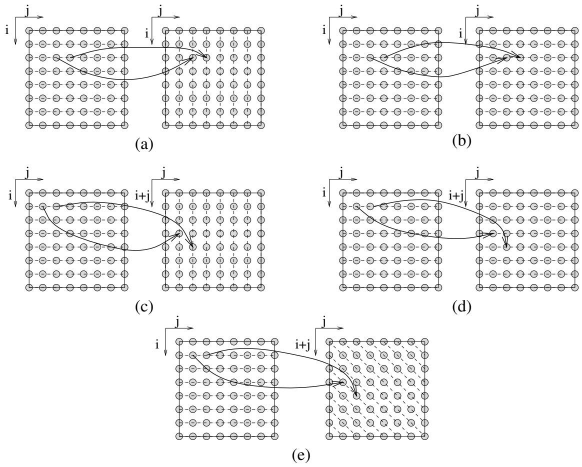

Figure 1: Different access patterns on two-dimensional iteration and data spaces. Two consecutive iterations access (a) two different columns; (b) the same row; (c) the elements in a diagonal (different columns); (d) the elements in a diagonal (different rows); (e) two consecutive elements in the same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

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 [19, 12]. 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 $A = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix}$ and $\bar{o} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$. In general, for a reference to an m-dimensional array inside an n-dimensional loop nest, the access matrix is $m \times n$ 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. [6].

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 that contains some sort of cache memory hierarchy, *cache locality* should be exploited. That is, a datum brought into the cache should be reused as much as possible before it is replaced. The reuse of the same data while it is still in the cache is termed as *temporal locality*, whereas the use of the nearby data in a cache line is called *spatial locality* [19]. We 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 two loops will improve the spatial locality for array X, but destroy the spatial locality for array W. Alternatively, if array X is stored in memory as row-major, without loop interchange the spatial locality will be exploited for both W and X. Figure 1(a) shows the original access pattern assuming a column-major layout for array X. The rectangular shape on the left denotes iteration space whereas that on the right denotes data space. The data accessed by two consecutive iterations fall into different columns, which causes the spatial locality to be poor for this reference. Instead, as shown in Figure 1(b), converting the memory layout of this array into row-major causes two consecutive iterations to access the same row. The situation for array Y, however, is more complicated, as neither a column-major (Figure 1(c)) nor a row-major (Figure 1(d)) layout storage improves locality. Instead, this array should be stored diagonally in memory so that two consecutive iterations access elements on the same diagonal as shown in Figure 1(e).

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

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

Under this definition, for a two-dimensional iteration space given by (i, j) and bounded by $1 \le i \le 20$ and $1 \le j \le 20$, 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.

3 Overview of Hyperplanes

In an m-dimensional space, a *hyperplane* can be defined as a set of tuples $(a_1, a_2, ..., a_m)$ such that $g_1 a_1 + g_2 a_2 + ... + g_m a_m = c$, where g_1, g_2,...,g_m are rational numbers called hyperplane coefficients and c is a rational number called hyperplane constant [7, 16]. A hyperplane vector $(g_1, g_2, ..., g_m)$ defines a hyperplane family where each member hyperplane has the same hyperplane vector but a different c value. For convenience, we use a row vector $\bar{g}^T = (g_1, g_2, ..., g_m)$ to denote such a hyperplane family whereas $\bar{g}$ corresponds to the column vector representation of the same hyperplane family. When there is no confusion we use g^T instead of $\bar{g}^T$.

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

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

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

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

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

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

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

As an example, let us focus on hyperplane family defined by $g^T = (0,1)$. Such a hyperplane family defines column hyperplanes on a two-dimensional 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 three- or higher-dimensional cases, we may have two take into account more than one hyperplane families. For example, 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 that have proximity in time access data points (both using R) that have proximity in space.*

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

It should be noted that our definition of spatial locality is broader than the usual meaning of the word as used by previous researchers. We also note that our spatial locality definition is only with respect to the innermost loop in the nest. Finally, we only consider self-spatial reuses (i.e., reuses that originate from a single reference). If a spatial reuse originates from distinct references, we call it group-spatial reuse [19]. Since, the cases where group-spatial reuse introduces an added dimension to the self-spatial reuse vector space are very rare, in this paper, we focus only on self-spatial reuses.

4 Optimizing Spatial Locality Using Data Layout Transformations

4.1 Problem Definition

We divide the problem of optimizing locality into two separate sub-problems:

- Determination of the optimal memory layouts that are defined by hyperplanes; and
- Data space transformations to obtain (or implement) the optimal layouts.

Each sub-problem can be solved independently. Previously, [4], [8], [9] and [11] offered algorithms to handle the first sub-problem whereas [15] and [11] 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.

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. 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.

The problem we address in this paper is defined as follows: "Given a program in which a number of arrays are accessed, what are the suitable memory layouts for each array such that the loop nests in the program will have spatial locality (as defined earlier) with respect to each reference that they enclose? If this is not possible, we want to maximize the number of references for which this is possible." As indicated in [3], due to some conditions related to storage and sequence assumptions about the arrays and to passing arrays as subroutine arguments, data transformations may not always be legal. We assume no such situation occurs for the example programs given in this paper.

4.2 Determining the Optimal Layouts

Let us now concentrate on two consecutive iterations $\bar{I}$ and $\bar{I}_{next}$ 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., $\bar{I} = (i_1, \cdots, i_{n-1}, i_n)^T$ and $\bar{I}_{next} = (i_1, \cdots, i_{n-1}, 1 + i_n)^T$. 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,*

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

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

Proof of this and the other lemmas in this paper can be found in [10]. 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 $A_U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $A_V = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, $A_W = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, $A_X = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, 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 [8] and [4] 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.

We now consider the following loop nest that accesses three and two dimensional arrays; we will focus on V.

```
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 matrix for the reference to array V is $A_V = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$.
Consider data points (d_1,d_2,d_3) and (d'_1,d'_2,d'_3). From Lemma (1), $\bar{g_V} \in Ker\{(1,0,1)\} \Longrightarrow \bar{g_V}^T = (0,1,0)$ 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 *locality group* and the intersection of the constraints defines a smaller locality group. To have a good spatial locality, the elements in this smaller locality group should be stored in consecutive memory locations. This can be seen from Table 2 where the elements of the array V accessed for some representative iterations are shown. We note that each group in the table satisfies both $d_2 = d'_2$ and $d_1 - d_3 = d'_1 - d'_3$. For example, from the first group $V(1,1,2)$ and $V(2,1,3)$ should be stored in memory together as they satisfy both of the constraints. As long as the arrays are large enough, it is sufficient to store the elements that satisfy both of the constraints as a locality group, provided that these two constraints do not conflict with each other. The relative order of these locality groups (if desired) are determined by considering a larger locality group. In this example, $d_2 = d'_2$ denotes a larger locality group which, for example, contains $V(1,1,2)$ and $V(1,1,3)$. Similarly, the other constraint, $d_1 - d_3 = d'_1 - d'_3$ also determines a locality group which, for example, includes $V(1,1,2)$ and $V(1,2,2)$. The choice between those locality groups is made by considering the second column of the access matrix which corresponds to the second innermost loop in the nest. We are now ready to give the following lemma.

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

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

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

$$\bar{g} \quad \in \quad Ker\{a_{n-k+1}\} \qquad (3)$$

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

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 $\bar{g_V} \in Ker\{(0,0,1)\} \Longrightarrow \bar{g_V}^T = (0,1,0)$ 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 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 that are in general difficult to optimize using 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 $A_{U_1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, and $A_{U_2} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$. Since, for optimizing spatial locality in the innermost loop, we are only interested in the last columns of the access matrices, and in this case both of the last columns are the same, there is no conflict between the two references, and the array can be stored in memory diagonally. Essentially, the observation is that as far as the data transformations are concerned, the imperfectly nested loops case is no different from the perfectly nested loops case as conflicts between references to the same array can occur in both cases.

An important point to note is that the access matrices for these two references are completely different from each other. That is, our approach, under specific conditions, can optimize an array that 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

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

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

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

4.6 Multiple Loop Nests

Notice that the ability to handle imperfectly nested loops within a unified framework also enables compiler to handle multiple loop nests in a single step. As proposed in [11], 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 $A_{U_1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, 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 $A_{U_1} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$, 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. In order to optimize spatial locality for the innermost loop, our approach focuses on the last columns of access matrices; thus, intuitively, if the Ker sets of the last columns of two references are *conformant,* (defined next) there will be no conflict in optimizing both.

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

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

Lemma 3 *Suppose an array U is accessed by s different references with access matrices $A_{U_1}, \ldots, A_{U_s}$. Let $a_{l_1}, \ldots, a_{l_s}$ 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 $Ker\{a_{l_1}\}, \cdots, Ker\{a_{l_s}\}$ are conformant.*

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.*

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 $A_{U_1} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $A_{U_2} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, $A_{U_3} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$, and $A_{U_4} = \begin{pmatrix} 1 & 1 \\ 0 & 2 \end{pmatrix}$. We note that $Ker\{a_{l_1}\}$, $Ker\{a_{l_2}\}$ and $Ker\{a_{l_3}\}$ are conformant but not $Ker\{a_{l_4}\}$. Therefore, the first three references can be optimized without conflict whereas there is a conflict with the fourth reference. In such a case, our method favors the largest subset of conflict-free references and decides a diagonal layout with the hyperplane $(1,-1)$ instead of $(2,-1)$. In the general case, however, whenever the conflicting references occur in different nesting levels, a *prioritizing* scheme is needed.

Investigation into such a scheme however is beyond the scope of this paper. Ideally, the references should be prioritized according to their frequency of access. Profile information might be useful for this purpose.

4.8 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 [18].

• Our data layout optimization framework generates a code such that (if possible) only the innermost loops carry spatial reuse.

To see why these help in a multiprocessor environment, it is sufficient to note that parallelizing a loop that carries spatial locality is one of the main reasons for false sharing, a phenomenon that occurs when two or more processors access logically separate data placed on the same cache line [5]. Since our framework causes spatial locality to be carried by the innermost loops, in most cases, the compiler can safely parallelize (provided the dependences allow it) the outermost loops without an apparent danger of false sharing.

In those situations where the compiler is able to parallelize only the innermost loops, the performance of data layout transformations in multiprocessors maybe rather poor.

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 [15] 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 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

$$L_{U_{default}} M_U = L_{U_{desired}}. \tag{4}$$

When this equation is solved for the elements of M_U, some of the elements of M_U (or some relations between them) will be determined. The unknown 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 can be found as follows: $L_{U_{desired}} = (1, 0)$, $L_{V_{desired}} = (0, 1)$, $L_{W_{desired}} = (0, 1)$, $L_{X_{desired}} = (1, -1)$, and $L_{Y_{desired}} = (1, 1)$. Using equation (4),

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

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

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

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

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

where $\times$ stands for an unknown entry. These matrices should be completed such that they should be nonsingular. Fortunately, the completion algorithms offered by [12] can be used for this purpose. An example solution is as follows. $M_U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, $M_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $M_W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $M_X = \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix}$, and $M_Y = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. From these matrices, compiler obtains the transformed access matrices as follows:

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

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. 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 should emphasize that our method can also transform the same original loop nest assuming that the default memory layout is row-major or diagonal (see [10]).

We now briefly discuss the problem of modifying the array declaration statements. Our approach uses the method of extreme values of affine functions first used by Banerjee [2] in dependence testing. Given an affine function of a number of variables and inequalities that represent the bounds for the variables, the extreme values method finds the maximum and minimum values of the affine function in the bounded region. The method applies to non-rectilinear regions as well (see [2]). In this method, the computed extreme values may not be tight as could have been if the Fourier-Motzkin elimination [17] had been used. However, for many programs, applying Fourier-Motzkin elimination to obtain tight bounds may be too expensive.

6 Experimental Results

We present performance results for eight programs: `matmult` is a matrix–multiplication routine with two versions (with and without loop unrolling)[1]; `syr2k` is banded matrix update routine from BLAS; `adi` is from Livermore kernels; `btrix`, `vpenta` and `cholesky`

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

are from Spec92/NASA benchmark suite; and `transpose` is a matrix transpose program from NWChem [14], a software package for computational chemistry. The `matmult`, `syr2k` and `transpose` programs consist 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 [12] 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.

We report execution times in seconds for up to 8 processors on the SGI Origin 2000, distributed–shared–memory machine. This machine 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.

The results on the SGI Origin 2000 are given in Figure 2. 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.

From our experiments, we observe that the data layout transformations are very effective in optimizing cache locality for uniprocessors. 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 `cholesky` and `btrix`).

7 Related Work

Loop transformations have been used for optimizing cache locality in several papers [12, 19, 13]. Results have shown that on several architectures the speedups achieved by loop transformations alone can be quite large.

Only a few papers have considered data transformations to optimize locality. O'Boyle and Knijnenburg [15] focus on restructuring the code given a data transformation matrix; though they show their method can be used for optimizing spatial locality. In comparison, we concentrate more on the problem of determining suitable layouts, derive a simple technique that can be applicable to multiple nests as well. Anderson, Amarasinghe and Lam [1] 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. While they restrict themselves to strip-mining and permutation only, we consider all types of layout transformations expressible by hyperplanes. Cierniak and Li [4] present a unified approach to optimize locality. Their approach employs both data space and iteration space transformations; but restricts search space for both types of transformations. Leung and Zahorjan [11] present an array restructuring framework to optimize locality. As against theirs, our technique is based on explicit representation of memory layouts. Finally, the previous work of the authors [8, 9] considers only dimension re-indexing and and like [4] uses a sort of exhaustive search to detect array layouts.

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 that has a default layout for all arrays. We mainly concentrated on the first subproblem. Experimental results on SGI Origin 2000 indicate that our framework can find opportunities for optimizing spatial locality without changing the access pattern of the loops. Our technique can be applied to any architecture with a memory hierarchy including uniprocessor machines.

References

[1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th SIGPLAN Symp. Prin. & Prac. Par. Prog.*,, July 1995.

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

[3] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. M. Anderson. Data-distribution support on distributed-shared memory multiprocessors. In *Proc. Programming Language Design and Implementation (PLDI)*, 1997.

[4] 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.

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

[6] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations, *Journal of Par. & Dist. Comp.*, 5:587–616, 1988.

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

[8] 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.

[9] M. Kandemir, J. Ramanujam, and A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines. In *Proc. PACT 97,* pages 236–247, 1997.

[10] 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.

[11] 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.

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

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

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

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

[16] J. Ramanujam, and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. In *IEEE Trans. Par. & Dist. Sys.*, 2(4):472–482, Oct. 1991.

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

[18] E. Torrie, C-W. Tseng, M. Martonosi, and M. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proc. PACT 95.*

[19] M. Wolf, and M. Lam. A data locality optimizing algorithm. In *Proc. SIGPLAN Conf. Prog. Lang. Des. & Impl.*, pages 30–44, 1991.

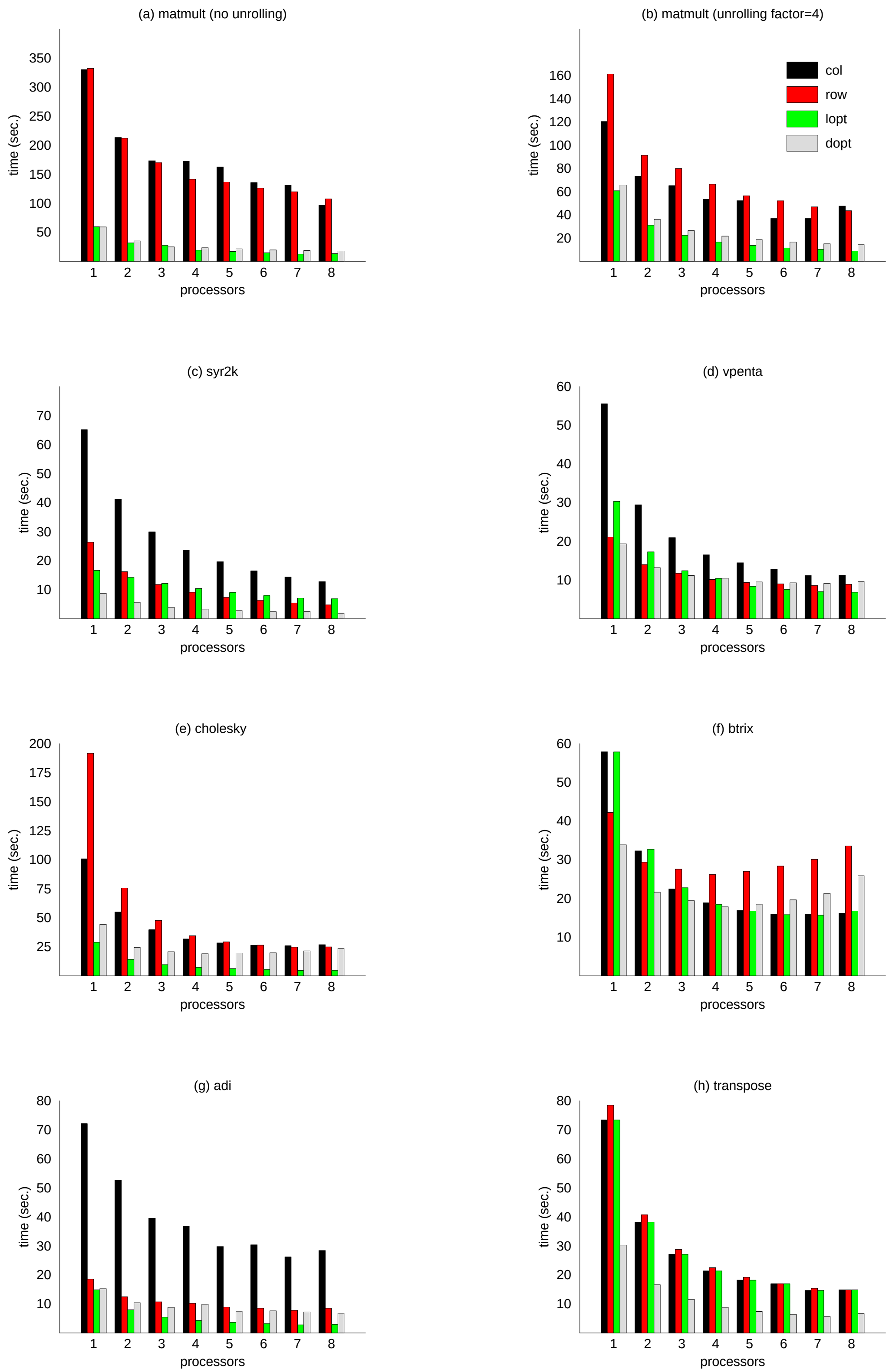

Figure 2: Execution times (in seconds) on the SGI Origin 2000. [The problem sizes (in doubles) are as follows: `matmult` and `matmult/u` – 1024×1024 matrices; `syr2k` – 1024×1024 matrices and $b = 300$; `vpenta` – $4 \times 720 \times 720$ 3D arrays and 720×720 2D arrays; `cholsky` – the size parameters 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].