A Layout-Conscious Iteration Space Transformation Technique

Mahmut Kandemir, *Member*, *IEEE*, J. Ramanujam, *Member*, *IEEE*,
Alok Choudhary, *Senior Member*, *IEEE*, and Prithviraj Banerjee, *Fellow*, *IEEE*

Abstract—Exploiting locality of references has become extremely important in realizing the potential performance of modern machines with deep memory hierarchies. The data access patterns of programs and the memory layouts of the accessed data sets play a critical role in determining the performance of applications running on these machines. This paper presents a cache locality optimization technique that can optimize a loop nest even if the arrays referenced have different layouts in memory. Such a capability is required for a global locality optimization framework that applies both loop and data transformations to a sequence of loop nests for optimizing locality. Our method uses a single linear algebra framework to represent both data layouts and loop transformations. It computes a nonsingular loop transformation matrix such that, in a given loop nest, data locality is exploited in the innermost loops, where it is most useful. The inverse of a nonsingular transformation matrix is built column-by-column, starting from the rightmost column. In addition, our approach can work in those cases where the data layouts of a subset of the referenced arrays is unknown; this is a key step in optimizing a sequence of loop nests and whole programs for locality. Experimental results on an SGI/Cray Origin 2000 nonuniform memory access multiprocessor machine show that our technique reduces execution times by as much as 70 percent.

Index Terms—Data reuse, cache locality, memory layouts, loop transformations, program optimization.

1 Introduction

As the disparity between processor and memory speeds continues to increase, most modern computer systems have resorted to the use of memory hierarchies with one or more levels of cache memory [11]. This results in a possible reduction of the average memory access times. The performance of programs on such systems can be significantly improved if they are written in such a way that a majority of memory references are satisfied by the cache. Manual performance improvement by restructuring code and data is tedious, error prone, and leads to nonportable code. Therefore, it is important that this task be left to an optimizing compiler [30]. Several automatic restructuring strategies have been proposed by compiler researchers, including loop (iteration space) as well as array layout (data space) transformations [30], [27], [39], [26], [19], [31]. The basic idea is to automatically modify the access pattern of a program such that the data (a single element or a cache block) brought into cache after a cache miss is reused as much as possible before being discarded from the cache. These techniques can be classified into three groups as described below.

Loop transformations: Consider an array reference $U[i, j]$ in a two-deep loop nest where the outer loop is i and the inner loop is j. Assuming that array U is stored in memory in column-major order (as in Fortran) and the trip count N of both loops is very large, successive iterations of the j-loop will touch different columns of array U that will very likely be mapped to different cache lines. Let us focus on a particular cache line that holds the initial part of a column. Before that column is accessed again, the j-loop sweeps through N different values, so, it is very likely that this cache line will be discarded from the cache before it is reused. Consequently, in the worst case, every access to U will involve a data transfer between the cache and main memory resulting in high access latencies. A solution to this problem is to *interchange* the loops i and j, making the i-loop innermost. As a result, a cache line brought into memory will be reused in a number of successive iterations of the i-loop, provided that the cache line is large enough to hold a number of array elements. Previous research in optimizing compilers [39], [27], [28], [30], [42] has resulted in several algorithms that detect and perform this loop interchange and other loop-based locality enhancing transformations automatically. These well-understood *loop transformations* can be used for optimizing temporal as well as spatial locality. But, data dependences in a nest may prevent the application of certain loop transformations [42]. Also, loop transformations are not very successful in optimizing locality in imperfectly nested [23] and explicitly parallelized loops [7].

Data transformations: Alternately, the locality problem can be tackled by a technique called *data* (*memory layout*) *transformation* or *array restructuring*. It is easy to see that if the memory layout of array U mentioned above is changed from column-major to row-major (without changing the

- *M. Kandemir is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802. E-mail: kandemir@cse.psu.edu.*
- *J. Ramanujam is with the Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. E-mail: jxr@ee.lsu.edu.*
- *A. Choudhary and P. Banerjee are with the Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. E-mail: {choudhar, banerjee}@ece.nwu.edu.*

Manuscript received 6 Apr. 1999; revised 19 June 2000; accepted 19 Oct. 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 109554.

```
do i = 1, N
 do j = 1, N
  U[i+j,j] = U[j,j] + U[j,i+j] + V[i,i+j]
 end do
end do

do i = 1, N
 do j = 1, N
  U[i,i-j] = V[i,i-j] + 1
 end do
end do
```

(a)

```
do u = 1, N
 do v = 1, N
  U[u+v,v] = U[v,v] + U[v,u+v] + V[u,u+v]
 end do
end do

do u = 1-N, N+1
 do v = max(1,1-u), min(N,N-u)
  U[u+v,v] = V[u,u+v] + 1
 end do
end do
```

(b)

Fig. 1. (a) The original code. (b) The transformed code. (Note: The transformed code exhibits good spatial locality if array U has diagonal memory layout and array V is row-major.)

loop order), then successive iterations of the inner j-loop can reuse the data in the cache. Recently, several authors [31], [7], [26], [18], [19] have proposed techniques to determine optimal memory layouts automatically. Such data transformation techniques are promising because they are not constrained by data dependences and are applicable to imperfect nests and explicitly parallelized loops, but the effect of a layout transformation is *global*, meaning that it affects the cache behavior of all loop nests that access the array assuming a fixed layout for each array throughout the execution of the program. In this paper, we consider the possibility that different arrays in the program may have different layouts. Therefore, in the general case, given a loop nest, the compiler is faced with an optimization problem that involves finding an appropriate loop transformationm assuming that the arrays in the nest may have different—perhaps unspecified—layouts. Unfortunately, current loop transformation techniques are unable to handle this since they assume a *fixed* (column-major, as in Fortran, or row-major, as in C) memory layout for all arrays in the program. Another significant problem with data layout transformations is that they cannot optimize a given reference for temporal locality [7], [26].

Combining data and loop transformations: It seems reasonable to combine loop and data transformations, resulting in a framework that is more powerful than using just loop or just data transformations. To motivate the discussion, consider the program fragment in Fig. 1a, assuming that arrays U and V are stored column-major by default. The first loop nest accesses array U diagonally and array V by rows; the second loop nest accesses both arrays by rows. Due to data dependence and conflicting access patterns of the arrays, the first loop nest cannot be optimized for locality of both arrays using loop transformations alone. Using a data layout transformation framework, one can determine that array U should be diagonally stored in memory and array V should be row-major. Now, having optimized the first loop nest, we focus on the second. A quick analysis shows that, for the best locality, either the loops should be interchanged and the arrays should be stored diagonally in memory or both the arrays should have row-major layouts without the need for any loop transformations. We note that neither of these solutions is satisfactory. The reason is that the layout of U is fixed as diagonal and that of array V as row-major in the first loop nest. Since we do not consider dynamic layout changes, accesses to one of the arrays in the second loop nest is unoptimized in both the solutions discussed. What we need for this second loop nest is a loop transformation that optimizes both the references under the assumption that the memory layout of the associated arrays are distinct: One of them is diagonal and the other one is row-major, as found in the first nest. This is the main problem addressed in this paper and we present a framework, using which such a loop transformation can be derived. The resulting code is shown in Fig. 1b. Notice that this program exhibits very good locality if array U is stored diagonally and V has a row-major layout. Notice also that this optimized code requires an additional data transformation step in a compiler that assumes a default (canonical) memory layout for all arrays. This last transformation step is rather mechanical; the associated details are beyond the scope of this paper and can be found elsewhere [26], [15], [31].

When data transformations are used in a program that involves multiple loop nests, in order to select suitable memory layouts, an optimizing compiler needs to consider multiple loop nests before making a decision. The key point here is that when we determine the layout of an array in a loop nest, we need to propagate this new layout information to the other nests in which the same array is accessed. There is a distinct possibility that this new layout will not be suitable (let alone optimal) for some of these other loop nests [26], [15], [31], [32].

In this paper, we present a framework that, given a loop nest, derives a loop transformation for the case where distinct arrays accessed in the nest may have *different* memory layouts. In addition, our solution works in the presence of unspecified (or undetermined) layouts for a subset of the arrays referenced; this is of particular interest in enhancing locality beyond a single nest in a global locality optimization problem involving multiple nests. The framework subsumes previous iteration space-based locality-enhancing *linear* transformation techniques that assume a *fixed* memory layout for all arrays. Our solution can be used as part of a larger framework that restructures array layouts globally for multiple loop nests.

Outline: The remainder of this paper is organized as follows: In Section 2, we present a brief review of the necessary technical background, followed by our framework used for a mathematical representation of the memory

layout information. Section 3 presents a loop transformation framework assuming that the memory layout for all arrays is column-major. In Section 4, we generalize our approach to attack the problem of optimizing a loop nest, assuming that the arrays referenced may have distinct memory layouts. We give our experimental results, obtained on an eight-processor SGI/Cray Origin 2000 nonuniform memory access multiprocessor, in Section 5. We review related work on data locality in Section 6 and conclude the paper in Section 7.

2 Background

In this paper, we consider nested loops. An iteration in an n-nested loop is denoted by an *iteration vector* $\bar{I} = [i_1, i_2, \cdots, i_n]^T$. Note that we write column vectors as a transpose of row vectors. We assume that the array subscript expressions and loop bounds are *affine functions* of enclosing loop indices and symbolic constants. In such a loop nest, each reference to an m-dimensional array U can be modeled by an *access* (or *reference*) *matrix* $\mathcal{L}$ of size $m \times n$ and an m-dimensional *offset vector* $\bar{o}$ [27], [39], [42]. For example, a reference $U[i_1 + i_2, i_2 + 1]$ in a loop nest of depth two can be represented by $\mathcal{L}\bar{I} + \bar{o}$, where

$$\mathcal{L} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

and $\bar{o} = [0, 1]^T$. We focus on loop transformations that can be represented by integer nonsingular $n \times n$ matrices. The effect of such a transformation $\mathcal{T}$ is that each iteration vector $\bar{I}$ in the original loop nest is transformed to $\mathcal{T}\bar{I}$. Therefore, loop bounds and subscript expressions should be modified accordingly. Let $\bar{I}' = \mathcal{T}\bar{I}$. Since $\mathcal{T}$ is invertible, the transformed reference can be written as $\mathcal{L}\bar{I} + \bar{o} = \mathcal{L}\mathcal{T}^{-1}\bar{I}' + \bar{o}$. The new loop bounds are computed using Fourier-Motzkin elimination. An iteration space transformation is legal if it preserves all data dependences in the original loop nest [42]. A linear transformation, represented by $\mathcal{T}$, is legal if, after the transformation, $\mathcal{T}\bar{d}$ is lexicographically nonnegative for each data dependence vector $\bar{d}$ in the original nest. The approach presented in this paper attempts to determine a $\mathcal{T}$ matrix for a given loop nest. The uniqueness of our approach is that (unlike previous loop transformation techniques) it determines $\mathcal{T}$ even if the arrays referenced in the nest have different memory layouts.

2.1 Memory Layout Representation Using Hyperplanes

In a previous paper [15], we presented a framework that represents memory layouts of arrays using hyperplanes. In this section, we briefly review the concepts presented there. A *hyperplane* defines a set of elements $[\jmath_1, \cdots, \jmath_m]^T$ that satisfies the equation $g_1\jmath_1 + g_2\jmath_2 + \cdots + g_m\jmath_m = c$ for a constant c. Here, $g_1, \cdots, g_m$ are rational numbers called hyperplane coefficients and c is a rational number called the hyperplane constant [33]. The hyperplane coefficients can be written collectively as a hyperplane vector $\bar{g} = [g_1, \cdots, g_m]^T$. Where there is no confusion, we omit the transpose. A *hyperplane family* is a set of hyperplanes defined by $\bar{g}$ for different values of c. It can be used to *partially* represent the memory layout of an array. We assume that the array elements on a specific hyperplane are stored in consecutive memory locations. Thus, for an array whose memory layout is column-major, each column represents a hyperplane whose elements are stored in memory consecutively. Given a large array, the relative storage order of the hyperplanes with respect to each other may not be important. Consequently, the hyperplane vector $[1, 0]^T$ denotes a row-major layout, $[0, 1]^T$ denotes column-major layout, and $[1, -1]^T$ defines a diagonal layout, all for two-dimensional arrays. Two array elements, $\bar{J}$ and $\bar{J}'$, belong to the same hyperplane $\bar{g}$ if the *locality constraint* $\bar{g}^T\bar{J} = \bar{g}^T\bar{J}'$ holds. For example, in a two-dimensional array stored as row-major, array elements $[5, 4]^T$ and $[5, 9]^T$ belong to the same hyperplane (i.e., the same row), but elements $[5, 4]^T$ and $[6, 4]^T$ do not. We say that two array elements which belong to the same hyperplane have *spatial locality*. Although this definition of spatial locality is somewhat coarse and does not hold at the array boundaries, it is suitable for our locality optimization strategy.

In a two-dimensional array space, a single hyperplane family is sufficient to partially define a memory layout. In higher dimensions, however, we may need more hyperplane families. Consider a three-dimensional array U whose layout is column-major. Such a layout can be represented using two hyperplanes: $\bar{g} = [0, 0, 1]^T$ and $\bar{h} = [0, 1, 0]^T$. We can write these two hyperplanes collectively as a *layout constraint matrix* or, simply, a *layout matrix*:

$$L_U = \begin{bmatrix} \bar{g}^T \\ \bar{h}^T \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

In such a case, two data elements $\bar{J}$, and $\bar{J}'$, are said to have spatial locality if the locality constraints $\bar{g}^T\bar{J} = \bar{g}^T\bar{J}'$ and $\bar{h}^T\bar{J} = \bar{h}^T\bar{J}'$ hold. The elements that have spatial locality should be stored in consecutive memory locations. This idea can easily be generalized to higher dimensions. Unless stated otherwise, we assume a column-major memory layout for all arrays. In Section 4, we show how to generalize our technique to optimize a given loop nest where a number of arrays with *different* memory layouts are accessed. Our optimization technique can work with any memory layout that can be represented using hyperplanes. In general, the column-major layout for an m-dimensional array can be represented by an $(m-1) \times m$ matrix $L = [l_{ij}]$, where $l_{i(m-i+1)} = 1$ for $1 \leq i \leq (m-1)$ and $l_{ij} = 0$ for the remaining elements. A thorough discussion of matrix-based layout representation can be found elsewhere [15].

3 Transformations for the Single Layout Case

3.1 Optimizing Spatial Locality

Our goal is to transform a loop nest such that spatial locality will be exploited in the inner loops in the transformed nest. That is, when we transform a loop nest, we want two consecutive iterations of the innermost loop to access array elements that have spatial locality. In particular, wherever possible, we want the accessed array elements to be next to each other so that they can be in the same cache line (or

neighboring cache lines). This can be achieved if, in the transformed loop nest, the elements accessed by consecutive iterations of the innermost loop satisfy appropriate locality constraints discussed in the previous section.

Let $\mathcal{Q} = \mathcal{T}^{-1}$ for convenience. Ignoring the offset vector (as it does not have any effect on self-spatial locality), after transformation $\mathcal{T}$, the new iteration vector $\bar{I}$ accesses (through $\mathcal{L}$) the array element $\mathcal{L}\mathcal{Q}\bar{I}$.

We first focus on two-dimensional arrays. For such an array U, the layout constraint matrix for the *column-major* layout is $L_U = [0, 1]$. Except for iteration space boundaries, two consecutive iteration vectors can be written as[1] $\bar{I} = [\imath_1, \imath_2, \cdots, \imath_{n-1}, \imath_n]^T$ and $\bar{I}_{next} = [\imath_1, \imath_2, \cdots, \imath_{n-1}, 1 + \imath_n]^T$. The data elements accessed by $\bar{I}$ and $\bar{I}_{next}$ through a reference represented by access matrix $\mathcal{L}$ will have spatial locality if

$$[0,1]\mathcal{L}\mathcal{Q}\bar{I} = [0,1]\mathcal{L}\mathcal{Q}\bar{I}_{next} \quad \Rightarrow \quad [0,1]\mathcal{L}\mathcal{Q}[0,\cdots,0,1]^T = 0$$
$$\Rightarrow \quad \bar{l}_2\bar{q}_n = 0 \quad \Rightarrow \quad \bar{q}_n \in Ker\{\bar{l}_2\},$$

where $\bar{l}_2$ and $\bar{q}_n$ are the last row of matrix $\mathcal{L}$ and last column of matrix $\mathcal{Q}$, respectively. Since $\bar{l}_2$ is known, we can always choose $\bar{q}_n$ from its null set (Ker set).

Notice that this technique determines only the last column of the matrix $\mathcal{Q}$. The remaining elements can be filled in any way as long as the resulting matrix $\mathcal{Q}$ is nonsingular and its inverse ($\mathcal{T}$) does not violate any data dependence. Later, we will focus on how to complete a partially filled $\mathcal{Q}$ matrix. For now, to see why the last column of $\mathcal{Q}$ is so important for locality and to obtain the *generalization* of this fixed-layout based model for higher-dimensional arrays, let us consider a reference to an m-dimensional array in an n-dimensional loop nest. Assume that $\mathcal{L} = [\,\bar{l}_1 \;\; \bar{l}_2 \;\; \cdots \;\; \bar{l}_m\,]^T$ and $\mathcal{Q} = [\,\bar{q}_1 \;\; \bar{q}_2 \;\; \cdots \;\; \bar{q}_n\,]$, where $\bar{l}_j$ is the jth row of $\mathcal{L}$ and $\bar{q}_k$ is the kth column of $\mathcal{Q}$. Assuming $i_1, i_2, \cdots, i_n$ are the loops in the nest after the transformation, omitting the offset vector, the new reference matrix is

$$\mathcal{L}\mathcal{Q}[i_1, \cdots, i_n]^T =$$
$$[\bar{l}_1\bar{q}_1 i_1 + \cdots + \bar{l}_1\bar{q}_n i_n, \cdots, \bar{l}_m\bar{q}_1 i_1 + \cdots + \bar{l}_m\bar{q}_n i_n].$$

Since the spatial locality behavior of a reference is mainly determined by the innermost loop (in our case, i_n) and all $\bar{l}_j$ are known, $\bar{q}_n$ is the sole factor that determines spatial locality. Our objective is to select $\bar{q}_n$ such that $\bar{l}_j\bar{q}_n$ will be 0 for each j, where $2 \leq j \leq n$ and $\bar{l}_1\bar{q}_n$ will be a small integer constant. In the two-dimensional case, since $m = 2$, selecting $\bar{q}_n$ from $Ker\{\bar{l}_2\}$ achieves precisely this goal. Using a similar reasoning, we can see that, in higher-dimensional cases, for good spatial locality in the innermost loop, the following relations should be satisfied:

$$\bar{q}_n \in Ker\{\bar{l}_2\}, \bar{q}_n \in Ker\{\bar{l}_3\}, \cdots, \bar{q}_n \in Ker\{\bar{l}_m\}.$$

To illustrate this, we consider the program fragment shown in Fig. 2a. The access matrices are

$$\mathcal{L}_C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathcal{L}_A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$
$$\text{and } \mathcal{L}_B = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

We first use $\bar{l}_3$ for each array:

$$\text{For array } C: \quad \bar{q}_3 \in Ker\{[0,0,1]\} \quad \Rightarrow \quad \bar{q}_3 = [\times, \times, 0]^T;$$
$$\text{For array } A: \quad \bar{q}_3 \in Ker\{[0,1,0]\} \quad \Rightarrow \quad \bar{q}_3 = [\times, 0, \times]^T;$$
$$\text{For array } B: \quad \bar{q}_3 \in Ker\{[1,0,0]\} \quad \Rightarrow \quad \bar{q}_3 = [0, \times, \times]^T.$$

These three equations, together, imply that $q_{13} = q_{23} = q_{33} = 0$; in other words, $\bar{q}_3$ should be the zero vector. Since we cannot have a zero column in the inverse of a transformation matrix (as the transformation matrix should be nonsingular by definition), we have to *eliminate* one of the equations and then reattempt to find a solution.

Our elimination scheme is based on profile information collected. With each reference in the nest, we associate a *weight* that represents the number of times this reference is touched in a typical execution. Note that, in our formulation, we have an equation per reference.[2] In cases where we need to eliminate an equation, we select the one whose associated reference has the minimum weight among all. Obviously, this strategy tries to minimize the runtime impact of an equation not taken into account. In our current example, since the references to arrays A and B have the same weight, we can select either and eliminate its equation. Note that, in case a single elimination does not lead to a solution, we continue by eliminating the equation of reference with the next minimum weight and so on. Returning to our example, if we eliminate the equation on B, we have $q_{23} = q_{33} = 0$. We now focus on $\bar{l}_2$ (middle row) from each reference matrix:

$$\text{For array } C: \quad \bar{q}_3 \in Ker\{[0,1,0]\} \quad \Rightarrow \quad \bar{q}_3 = [\times, 0, \times]^T;$$
$$\text{For array } A: \quad \bar{q}_3 \in Ker\{[0,0,1]\} \quad \Rightarrow \quad \bar{q}_3 = [\times, \times, 0]^T;$$
$$\text{For array } B: \quad \bar{q}_3 \in Ker\{[0,1,0]\} \quad \Rightarrow \quad \bar{q}_3 = [\times, 0, \times]^T.$$

From these three relations, we have $q_{23} = q_{33} = 0$, which is consistent with the previous findings. We now have a partially filled matrix

$$Q = \begin{bmatrix} \times & \times & \times \\ \times & \times & 0 \\ \times & \times & 0 \end{bmatrix}.$$

We consider only the two permutation matrices:[3]

$$\mathcal{Q}_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

1. The following formulation will be based on this assumption that two consecutive iteration vectors do not fall into an iteration space boundary. This is reasonable as, in a nest that consists of loops with large trip counts, the majority of the consecutive iteration vector pairs do not fall into boundaries.

2. In the current example, the two references to array C have the same equation, so, they can be considered as a single reference with twice the weight.

3. Notice that our approach in general can result in any nonsingular transformation matrix. Here, in order to keep the presentation simple, we use only permutation matrices. In the rest of the paper, we will focus on unimodular matrices explicitly.

```
do i = 2, N
 do j = 1, N
  do k = 1, N-1
   C[i,j,k]=A[i,k,j]+B[k,j,i]+C[i-1,j-1,k+1]
  end do
 end do
end do
```

(a)

```
do u = 1, N
 do v = 1, N-1
  do w = 2, N
   C[w,u,v]=A[w,v,u]+B[v,u,w]+C[w-1,u-1,v+1]
  end do
 end do
end do
```

(b)

Fig. 2. (a) Original loop nest. (b) Transformed loop nest.

and

$$\mathcal{Q}_2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

which result in

$$\mathcal{T}_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

and

$$\mathcal{T}_2 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix},$$

respectively. Notice that the original loop nest has a data dependence vector $\bar{d} = [1, 1, -1]^T$ due to array C. Since $\mathcal{T}_1\bar{d}$ is lexicographically negative, this dependence vector renders $\mathcal{T}_1$ illegal, leaving only $\mathcal{T}_2$ as a legal permutation matrix. The transformed code is shown in Fig. 2b. Here, spatial locality is very good for arrays C and A, whereas spatial locality for array B is exploited only by the middle loop. This is the result of eliminating the equation associated with B.

3.2 Exploiting Temporal Locality

An important issue now is to integrate the handling of temporal locality into our framework. It is well-known that, as far as the innermost loops are concerned, optimizing temporal locality is more important than optimizing spatial locality because, if the innermost loop carries temporal reuse for a given reference, that reference can be kept in a register throughout the execution of the innermost loop (provided that there is no aliasing involving that reference). If, however, the reference in question can only be optimized for spatial locality, misses at cache line boundaries will be inevitable.

It is relatively easy to extend our approach to handle temporal reuse. Recall that, for spatial locality in the innermost loop, the following conditions should hold:

$$\bar{l_1}\bar{q_n} = y, \bar{q_n} \in Ker\{\bar{l_2}\}, \bar{q_n} \in Ker\{\bar{l_3}\}, \cdots, \bar{q_n} \in Ker\{\bar{l_m}\},$$

where y is a small integer value. For temporal locality, however, we should have the following:

$$\bar{q_n} \in Ker\{\bar{l_1}\}, \bar{q_n} \in Ker\{\bar{l_2}\},$$
$$\bar{q_n} \in Ker\{\bar{l_3}\}, \cdots, \bar{q_n} \in Ker\{\bar{l_m}\}.$$

That is, $\bar{q_n}$ should belong to the null space of all the rows of the access matrix.

We illustrate the approach for optimizing temporal locality using the example nest given in Fig. 3a. The access matrices are

$$\mathcal{L}_U = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \text{ and } \mathcal{L}_V = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

The data locality theory developed by Wolf and Lam [39] shows that it is not possible to optimize both the references for temporal locality. Therefore, let us attempt to optimize the reference to array U for temporal locality and the reference to array V for spatial locality. For the temporal locality of array U, $\bar{q_3} \in Ker\{\mathcal{L}_U\}$, i.e.,

$$\bar{q_3} \in Ker\left\{\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}\right\}.$$

To ensure spatial locality of array V, on the other hand, $\bar{q_3} \in Ker\{[1, 0, 0]\}$ should be satisfied. The vector $\bar{q_3} = [0, 1, -1]^T$ satisfies both conditions. With $\bar{q_3}$ as the last column, we can complete T^{-1} as

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}.$$

```
do i = 1, N
 do j = 1, N
  do k = 1, N
   U[j+k,i]=V[i+k,k]
  end do
 end do
end do
```

(a)

```
do u = 1, N
 do v = 2, 2N
  do w = max(-N+v,1), min(v-1,N)
   U[v,u]=V[u+v-w,u]
  end do
 end do
end do
```

(b)

Fig. 3. (a) Original loop nest. (b) Transformed loop nest.

The resulting program is shown in Fig. 3b. Note that temporal locality is exploited for array U and spatial locality is obtained for array V, both in the innermost loop.

An important problem in optimizing temporal locality is to select the reference(s) for which temporal locality can be optimized. For instance, in the example above, we could have easily tried to optimize array V for temporal locality instead of array U. Fortunately, Wolf and Lam [39] tell us how to check whether we can achieve temporal locality for a given reference. The idea is to compute the kernel set of the access matrix of the reference in question: If the kernel set is empty, we cannot achieve temporal locality; otherwise, at least in principle, temporal locality might be possible for that reference.

Our current approach to select the references to be optimized for temporal locality is rather straightforward. Simply put, we try to improve temporal locality for as many references as possible. For example, suppose that, in a given loop nest that has four references, only three of the references can be optimized for temporal locality. Then, we build our equations such that these three references are to be optimized for temporal locality and the remaining reference for spatial locality. If we cannot achieve this, then (instead of starting to eliminate the equations right away) we optimize one of these three references for spatial locality instead. We try to accomplish this by trying all three combinations in turn, where two references (out of three with potential temporal locality) are optimized for temporal locality. If any of these combinations leads to a solution, we stop. If all of them fail, then we can only achieve temporal locality for a single reference, in which case, we try to optimize each of these three references for temporal locality in turn.

3.3 Locality in Outer Loops

So far, we have only concentrated on determining the elements of the last column of $\mathcal{Q}$. While, for most cases, this is sufficient to improve locality, in situations where the trip count of the innermost loop is small and where some references exhibit temporal locality in the innermost loop, we may need to pay attention to the spatial locality carried by the outer loops as well. Let us focus on the spatial locality in the second innermost loop. This corresponds to determining the elements of the second rightmost column of $\mathcal{Q}$. We define $\bar{I}_k = [\imath_1, \cdots, \imath_{k-1}, \imath_k, \imath_{k+1}, \cdots, \imath_n]^T$ and $\bar{I}'_k = [\imath_1, \cdots, \imath_{k-1}, 1 + \imath_k, \imath_{k+1}, \cdots, \imath_n]^T$. This means that iteration vectors $\bar{I}_k$ and $\bar{I}'_k$ have exactly the same values for all loop index positions except the kth index, where they differ by one. In this case, we can exploit the spatial locality in the second innermost loop if

$$[0,1]\mathcal{L}\mathcal{Q}\bar{I}_{n-1} = [0,1]\mathcal{L}\mathcal{Q}\bar{I}'_{n-1} \text{ or } [0,1]\mathcal{L}\mathcal{Q}[0,\cdots,0,1,0]^T = 0,$$
$$\text{i.e., } \bar{l_m}\bar{q_{n-1}} = 0 \quad \Rightarrow \quad \bar{q_{n-1}} \in Ker\{\bar{l_m}\}.$$

4 Transformations for the Multiple Layout Case

In this section, we present the formulation for the most general case, where a number of arrays with possibly *different* memory layouts are referenced in a given loop nest. Our goal is to find a transformation matrix $\mathcal{T}$ such that locality is good for as many references as possible. We use the following notation:

- ν': the number of distinct references.
- R_σ: the reference σ where $1 \leq \sigma \leq \nu'$.
- $L_\sigma = [a^\sigma_{ij}]$: the $(m-1) \times m$ layout matrix for the m-dimensional array associated with R_σ.
- $\mathcal{L}_\sigma = [l^\sigma_{ij}]$: the $m \times n$ access (reference) matrix for R_σ.

In order to resolve conflicts arising from different references, we assume that, prior to our analysis, the references are ordered according to their relative importance. Without loss of generality, let the references be ordered as $R_1, \cdots, R_{\nu'}$, where R_1 is the most important reference and $R_{\nu'}$ is the least important. Ideally, the references should be ordered according to their access frequencies. But, in practice, unknown loop bounds, complex subscript expressions and conditional control flows make it difficult to estimate at compile time the number of times a reference will be executed. Therefore, we currently use profiling to determine the number of times each reference will be touched in a typical execution. As explained earlier, for each reference, we compute a weight and order the references according to their weight values.

Let us now focus on a single reference R_σ. Assuming $\bar{I}$ and $\bar{I}_{next}$ are two consecutive iteration vectors after applying $\mathcal{T} = \mathcal{Q}^{-1}$, the two data elements accessed by these iteration vectors through R_σ will have spatial locality if $L_\sigma\mathcal{L}_\sigma\mathcal{Q}\bar{I} = L_\sigma\mathcal{L}_\sigma\mathcal{Q}\bar{I}_{next}$ or $L_\sigma\mathcal{L}_\sigma\bar{q_n} = \bar{0}$, where $\bar{q_n}$ is the last column of $\mathcal{Q}$. On expanding this constraint, we derive the relation

$$\begin{bmatrix} a^\sigma_{11} & a^\sigma_{12} & \cdots & a^\sigma_{1m} \\ a^\sigma_{21} & a^\sigma_{22} & \cdots & a^\sigma_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a^\sigma_{(m-1)1} & a^\sigma_{(m-1)2} & \cdots & a^\sigma_{(m-1)m} \end{bmatrix} \times \begin{bmatrix} l^\sigma_{11} & l^\sigma_{12} & \cdots & l^\sigma_{1n} \\ l^\sigma_{21} & l^\sigma_{22} & \cdots & l^\sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ l^\sigma_{m1} & l^\sigma_{m2} & \cdots & l^\sigma_{mn} \end{bmatrix} \begin{bmatrix} q_{1n} \\ q_{2n} \\ \vdots \\ q_{nn} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \quad (1)$$

Setting $b^\sigma_{ij} = \sum_{k=1}^{m} a^\sigma_{ik} l^\sigma_{kj}$ for $1 \leq i \leq m-1$ and $1 \leq j \leq n$, we can rewrite the matrix equation above as

$$\begin{bmatrix} b^\sigma_{11} & b^\sigma_{12} & \cdots & b^\sigma_{1n} \\ b^\sigma_{21} & b^\sigma_{22} & \cdots & b^\sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b^\sigma_{(m-1)1} & b^\sigma_{(m-1)2} & \cdots & b^\sigma_{(m-1)n} \end{bmatrix} \begin{bmatrix} q_{1n} \\ q_{2n} \\ \vdots \\ q_{nn} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

or, in compact form, as $B^\sigma\bar{q_n} = \bar{0}$, where $B^\sigma = [b^\sigma_{ij}]$. Then, the determination of the last column of $\mathcal{Q}$ can be expressed as the problem of finding a vector from the solution space of this homogeneous system.[4] Notice that this solution only takes care of the reference R_σ. It is possible that, for some σ, where $1 \leq \sigma \leq \nu'$, the equation $B^\sigma\bar{q_n} = \bar{0}$ has no nontrivial

4. Additionally, we might want to add the constraints on $\bar{q}_{n-1}$ in order to exploit the spatial reuse in the second innermost loop. We do not consider this any further in this paper.

Input: A loop nest with a number of references, layout matrices for each reference and the data dependence matrix.
Output: A non-singular loop transformation matrix that observes all data dependences.

Step 1. Form the following homogeneous system

$$\left\{B^1\bar{q_n} = \bar{0}, \cdots, B^\nu\bar{q_n} = \bar{0}\right\} \equiv B\bar{q_n} = \bar{0}$$

where $B^\sigma = L_\sigma\mathcal{L}_\sigma$ and $B^\sigma\bar{q_n} = \bar{0}$ has non-trivial solutions for $1 \leq \sigma \leq \nu$. Also note that the references are ordered as explained in the text in Section 4. Eliminate the redundant equations and let τ be the number of the remaining rows in B. If $\tau = 0$, then exit (leave the loop nest as is) returning $\mathcal{T}$ = identity matrix.

Step 2. Solve the system by row-echelon reduction. This is achieved by transforming the augmented matrix $[B|0]$ of the system to a matrix $[\mathcal{C}|0]$ in reduced row echelon form [42]. After the reduction, let r be the number of non-zero rows in $\mathcal{C}$, where $1 \leq r \leq \tau$.

Step 3. If $r \geq n$ the solution space has no basis. In that case, delete the equations corresponding to the last reference and repeat *Step 2* until a $r < n$ is found. Let $p = n - r$. Note that p is guaranteed to be at least one.

Step 4. Write the solution $\bar{x}$ as a linear combination of vectors $\bar{x_1}, \bar{x_2}, \cdots, \bar{x_p}$ with the corresponding coefficients $\alpha_1, \alpha_2, \cdots, \alpha_p$: $\bar{x} = \alpha_1\bar{x_1} + \alpha_2\bar{x_2} + \cdots + \alpha_p\bar{x_p}$

Step 5. If $\bar{q_n} = \bar{x}$ can be written as an integer multiple of a unit vector, then check if any of the permutation matrices $\mathcal{T}$ whose last row is $\bar{q_n}$ written as a row vector (see text for an explanation) results in $\mathcal{T}\bar{d}$ being lexicographically positive for all dependence vectors $\bar{d}$. If this is the case, then exit returning that matrix as the transformation matrix.

Step 6. If *Step 5* does not apply, check if $q_{nn} \neq 0$. If this is the case, construct $\mathcal{T}$ as in Equation 4. Check if for all dependence vectors $\bar{d} = (d_1, \cdots, d_n)$ $\mathcal{T}\bar{d}$ is lexicographically positive. Using $q_{kn} = \sum_{i=1}^{p} \alpha_i x_{ki}$, we get constraints on $\alpha_1, \alpha_2, \cdots, \alpha_p$. If a valid $\mathcal{T}$ exists, then return that as the transformation matrix.

Step 7. If it is not possible to find a $\mathcal{T}$ in the previous steps, then we use the completion method described by Bik and Wijshoff [5] or by Li [27] to complete $\bar{q_n}$ to a non-singular matrix Q such that all the dependences are satisfied by $\mathcal{T} = \mathcal{Q}^{-1}$. If one such $\mathcal{T}$ exists, return that as the transformation matrix.

Fig. 4. Algorithm for determining the transformation matrix (it is assumed that $q_{nn} \neq 0$).

solution, i.e., the only solution is $\bar{q_n} = \bar{0}$. We do *not* consider any such references further. Let ν ($\nu \leq \nu'$) be the number of references for which the equation $B^\sigma\bar{q_n} = \bar{0}$ has nontrivial solutions. In order to obtain a transformation that satisfies all ν references, we have to set up and solve the following system

$$B^1\bar{q_n} = \bar{0},\ B^2\bar{q_n} = \bar{0},\ \cdots,\ B^\nu\bar{q_n} = \bar{0}. \quad (2)$$

Given a large number of references, this homogeneous system may not have a nontrivial solution. In that case, we eliminate all the equations arising due to some reference σ from consideration (i.e., from the system) and repeat the process. Our approach is to find the largest k $(1 \leq k \leq \nu)$ such that the system

$$B^1\bar{q_n} = \bar{0},\ B^2\bar{q_n} = \bar{0},\ \cdots,\ B^k\bar{q_n} = \bar{0}. \quad (3)$$

has a nontrivial solution. Since the references are arranged in decreasing order of importance, this appears to be reasonable. Note that this is a greedy strategy and may be suboptimal in some cases. A full discussion of profiling and a weighted selection of the set of references to use in the system is beyond the scope of this paper. The complete algorithm is given in Fig. 4.

Our approach to the problem of finding $\mathcal{Q}$ is based on first attempting to find solutions using simple methods and then resorting to expensive ones only as needed. We now describe a simple method. A solution to this homogeneous system is of the form $\bar{q_n} = \bar{x} = \alpha_1\bar{x_1} + \alpha_2\bar{x_2} + \cdots + \alpha_p\bar{x_p}$, where $\alpha_1, \alpha_2, \cdots, \alpha_p$ are integers and $\bar{x_1}, \bar{x_2}, \cdots, \bar{x_p}$ is a set of linearly independent vectors.

The first case we handle is one where $\bar{q_n}$ is a unit vector, i.e., a vector in which exactly one element is one and the remaining are zero. For n-element vectors, $\bar{e_k}$ is defined as the kth unit vector if its kth element is one and the remaining elements are zero. A matrix consisting of any permutation of unit vectors $\bar{e_k}$ is called a permutation matrix. If $\bar{q_n} = \bar{e_k}$, we fill out the first $n-1$ columns of $\mathcal{Q}$ with some permutation of unit vectors $\bar{e_1}, \cdots, \bar{e_{k-1}}, \bar{e_{k+1}}, \bar{e_n}$. Since the inverse of a permutation matrix Q is its transpose, this is equivalent to filling out the matrix $\mathcal{T} = \mathcal{Q}^{-1}$ whose last row is $\bar{q_n^T}$. We pick any such $\mathcal{T}$ that satisfies all the dependence vectors and we are done. Note that we can add extra constraints on $\bar{q_{n-1}}$ in order to exploit the spatial reuse in the second innermost loop.

Suppose now that $\bar{q_n}$ is not a unit vector. We fill out $\mathcal{Q}$ such that it is of the form

$$\mathcal{Q} = \begin{bmatrix} 1 & 0 & \cdots & 0 & q_{1n} \\ 0 & 1 & \cdots & 0 & q_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & q_{(n-1)n} \\ 0 & 0 & \cdots & 0 & q_{nn} \end{bmatrix}.$$

Assuming $q_{nn} \neq 0$,

$$\mathcal{T} = \mathcal{Q}^{-1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & -\frac{q_{1n}}{q_{nn}} \\ 0 & 1 & \cdots & 0 & -\frac{q_{2n}}{q_{nn}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -\frac{q_{(n-1)n}}{q_{nn}} \\ 0 & 0 & \cdots & 0 & \frac{1}{q_{nn}} \end{bmatrix}. \quad (4)$$

We have assumed that $q_{nn} \neq 0$ in order to keep the discussion simple. Note that if $q_{nn} < 0$, then the vector $-\bar{q_n}$ is a solution and we use that. We will now discuss the case $q_{nn} > 0$ without loss of generality. With this assumption, we are able to derive a closed form expression for $\mathcal{T} = \mathcal{Q}^{-1}$ as shown in (4). If $q_{nn} = 0$, we use the completion method described by Bik and Wijshoff [5] or by Li [27] and construct a nonsingular Q. Note, however, that these methods do not provide a closed form expression for $\mathcal{T}$.

Recall that a solution to (3) is of the form

$$\bar{q_n} = [q_{1n}, q_{2n}, \cdots, q_{nn}]^T = \alpha_1 \bar{x_1} + \alpha_2 \bar{x_2} + \cdots + \alpha_p \bar{x_p}, \quad (5)$$

where $\alpha_1, \alpha_2, \cdots, \alpha_p$ are integers and $\bar{x_1}, \bar{x_2}, \cdots, \bar{x_p}$ is a set of linearly independent vectors. We will refer to the collection of vectors $\{\bar{x_1}, \bar{x_1}, \cdots, \bar{x_p}\}$ as the matrix X, where x_{ki} denotes the kth element of $\bar{x_i}$. We denote the components of the vector $\bar{q_n}$ as $(q_{1n}, q_{2n}, \cdots, q_{kn})$. Thus, one can write $q_{kn} = \sum_{i=1}^{p} \alpha_i x_{ki}$. Let $\mathcal{D}$ be the original dependence matrix and let $\bar{d} \in \mathcal{D}$ be a dependence vector, where $\bar{d} = [d_1, d_2, \cdots, d_{n-1}, d_n]^T$. After the transformation $\mathcal{T}$, using the closed form for $\mathcal{T}$ from (4), we have the transformed dependence vector as

$$\mathcal{T}\bar{d} = \left[d_1 - \frac{q_{1n}}{q_{nn}} d_n, d_2 - \frac{q_{2n}}{q_{nn}} d_n, \cdots, d_{n-1} - \frac{q_{(n-1)n}}{q_{nn}} d_n, \frac{d_n}{q_{nn}} \right]^T.$$

Thus, if $\mathcal{T}\bar{d}$ is lexicographically positive, then the transformation $\mathcal{T}$ is legal. We note that if d_n is equal to zero, then this resulting dependence vector is always legal, provided that $\bar{d}$ is legal to begin with.

Since we have assumed that $q_{nn} > 0$ without loss of generality, the transformed dependence vector can be written as

$$\left[\frac{d_1 q_{nn} - q_{1n} d_n}{q_{nn}}, \frac{d_2 q_{nn} - q_{2n} d_n}{q_{nn}}, \cdots, \frac{d_{n-1} q_{nn} - q_{(n-1)n} d_n}{q_{nn}}, \frac{d_n}{q_{nn}} \right]^T.$$

Since $q_{nn} > 0$, the transformed dependence is legal if $[d_1 q_{nn} - q_{1n} d_n, d_2 q_{nn} - q_{2n} d_n, \cdots, d_{n-1} q_{nn} - q_{(n-1)n} d_n, d_n]^T$ is legal. Since q_{nk} can be written is terms of α_i and $\bar{x_i}$, this translates into conditions on α_i. A sufficient condition is that every element of the vector $(q_{1n}, q_{2n}, \cdots, q_{(n-1)n}) \leq 0$. This is derived by reasoning about the level of the dependence $\bar{d}$ for each dependence vector. Thus, if there is a valid transformation, the parameters $\alpha_1, \alpha_2, \cdots, \alpha_p$ in (5) can be chosen such that $\mathcal{T}\bar{d}$ is lexicographically nonnegative [27]. A full discussion of the legality is beyond the scope of this paper and the reader is referred to the associated technical report [20].

4.1 Handling Temporal Locality

Note that the approach currently explained aims at obtaining good spatial locality for as many references as possible. In order to handle temporal locality, we can use Wolf and Lam's method [39] to determine the references that can be optimized for temporal locality and then try to maximize the number of references with temporal locality. The most important modification is made to (1). For a given reference to have temporal locality, this equation should be satisfied no matter what the memory layout is, that is, in this case, we can rewrite the said equation as

$$\begin{bmatrix} l_{11}^{\sigma} & l_{12}^{\sigma} & \cdots & l_{1n}^{\sigma} \\ l_{21}^{\sigma} & l_{22}^{\sigma} & \cdots & l_{2n}^{\sigma} \\ \vdots & \vdots & \ddots & \vdots \\ l_{m1}^{\sigma} & l_{m2}^{\sigma} & \cdots & l_{mn}^{\sigma} \end{bmatrix} \begin{bmatrix} q_{1n} \\ q_{2n} \\ \vdots \\ q_{nn} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \quad (6)$$

In other words, we set $b_{ij}^{\sigma} = l_{ij}^{\sigma}$ for all i and j. Note that this modified form needs to be used only for the references for which we would like to optimize temporal locality. It should also be noted that our approach to temporal locality as explained so far stops the algorithm when a solution with maximum references with temporal locality is found. Of course, it might be the case that another solution with the same number of references with temporal locality has better performance. A more aggressive approach can find all possible solutions with the maximum number of references (with temporal locality) and then select the best one among them, using a selection criterion. The advantage of this scheme is that (depending on the accuracy of the selection criterion) it has a potential for finding a better solution than our current scheme. The drawback is that it might increase the compilation time significantly. Our experiments revealed that just picking up the first solution found and stopping the algorithm generally results in the best solution without unduly increasing compile time.

4.2 Example

As an application of this algorithm, consider the matrix multiplication code shown below. We assume that arrays A and C are row-major, whereas array B is column-major.

```
do i = 1, N
  do j = 1, N
    do k = 1, N
      C[i,j] += A[i,k] * B[k,j]
    end do
  end do
end do
```

The equations in our homogeneous system are $[1,0,0][q_{13}, q_{23}, q_{33}]^T = 0$, $[1,0,0][q_{13}, q_{23}, q_{33}]^T = 0$, and $[0,1,0][q_{13}, q_{23}, q_{33}]^T = 0$, corresponding to references to arrays C, A, and B, respectively. A solution is $\bar{q_3} = (0,0,1)^T$. The partially filled matrix

TABLE 1
Loop orders for Different Layout Combinations in the Matrix-Multiplication Code

C-A-B	for sequential execution	for parallel execution
c-c-c	j-k-i	j-k-i
c-c-r	k-j-i	j-k-i
c-r-c	j-i-k	j-i-k
c-r-r	j-k-i	j-k-i
r-c-c	i-k-j	i-k-j
r-c-r	k-i-j	i-k-j
r-r-c	i-j-k	i-j-k
r-r-r	i-k-j	i-k-j

A triple x-y-z in the first column refers to memory layouts for arrays C, A, and B, respectively, where c means column-major and r means row-major.

$$Q = \begin{bmatrix} \times & \times & 0 \\ \times & \times & 0 \\ \times & \times & 1 \end{bmatrix},$$

completed as

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix};$$

thus,

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Therefore, given the mentioned memory layouts, the original loop order i-j-k (from outermost to innermost position) is the best loop order from the locality point of view.

Table 1 shows the loop orders detected by our algorithm for the matrix multiplication nest under all possible permutation-based layout combinations. The middle column gives the order for sequential (uniprocessor) execution, whereas the rightmost column gives that for parallel (multiprocessor) execution. The order for parallel execution differs from that for sequential execution when the outermost loop in the sequential order cannot be parallelized due to either data dependence contraints [4] or data decomposition contraints (among parallel processors) [1]. In this case, we can interchange this loop with the closest loop that does not carry any data dependence if it is legal to do so. Determining the suitability of a locality-optimized sequential program for a parallel architecture is beyond the scope of this paper and is definitely an important issue that needs to be revisited in the future.

4.3 Partial Layout Information

A direct generalization of the approach presented in the previous section is important for optimizing a loop nest, assuming that some of the arrays have fixed, but possibly different, layouts, whereas the remaining arrays have not been assigned memory layouts yet. In such cases, an optimizing compiler should derive the best loop transformation as well as the optimal memory layouts for the arrays that have no assigned layout yet. This situation arises frequently in the task of optimizing a sequence of loop nests when we reach a loop nest in which some of the arrays referenced have their layouts determined while the layouts of the remaining arrays is yet to be determined (see the next section). We handle this problem using the following two steps: 1) Find a loop transformation that satisfies the references to the arrays whose layouts have already been determined and 2) taking into account this loop transformation (from the previous step), determine the optimal layouts of the remaining arrays referenced in the nest. The first step is handled using the approach discussed so far in this paper. The second step is, on the other hand, an array restructuring problem discussed in [15] and [19]. More details on utilizing the partial layout information can be found in [20].

TABLE 2
Platform Used in the Experiments

Number of Processors	8
Processor	195 MHz MIPS R10000
Out-of-Order Execution	4 instructions
Functional Units	5
L1 Cache	32 KB split, 2-way associative
L2 Cache	4 MB unified, 2-way associative
L1 latency	2-3 cycles
L2 latency	8-10 cycles
Page Size	16 KB (default)
Compiler (cc)	MIPSPro 7.2
Operating System	IRIX 6.5

5 Experimental Results

In this section, we demonstrate how our iteration space transformation technique improves performance on an 8-processor SGI/Cray Origin 2000 nonuniform memory access multiprocessor. The important characteristics of our platform are given in Table 2. Our presentation is in two main parts. First, we evaluate the effectiveness of our approach using a set of nine programs assuming a *fixed* memory layout for all arrays. Then, we measure the improvements in execution time with *different* layout combinations.

We experimented with the following programs: ADI from Livermore kernels; matmult, the classical i-j-k matrix multiplication routine; cholesky from [27]; vpenta, btrix, and cholsky[5] from the Spec92/Nasa7 benchmark suite; syr2k from Blas; and, finally, fnorm and bnorm from Odepack, a collection of solvers for the initial value problem for systems of ordinary differential equations. We use the C versions of these programs[6] and the codes are transformed for locality by a source-to-source translator written using the Omega library [21]. The resulting optimized codes are then compiled by the native optimizing compiler using the -O2 option with all low-level optimizations. In the experiments performed on multiple processors, for each program, we have chosen the best

5. Different from cholesky; uses two three-dimensional arrays.
6. In converting the programs from Fortran to C, we maintained the original locality by transforming the array layouts as well. This simply corresponds to permuting array subscript expressions.

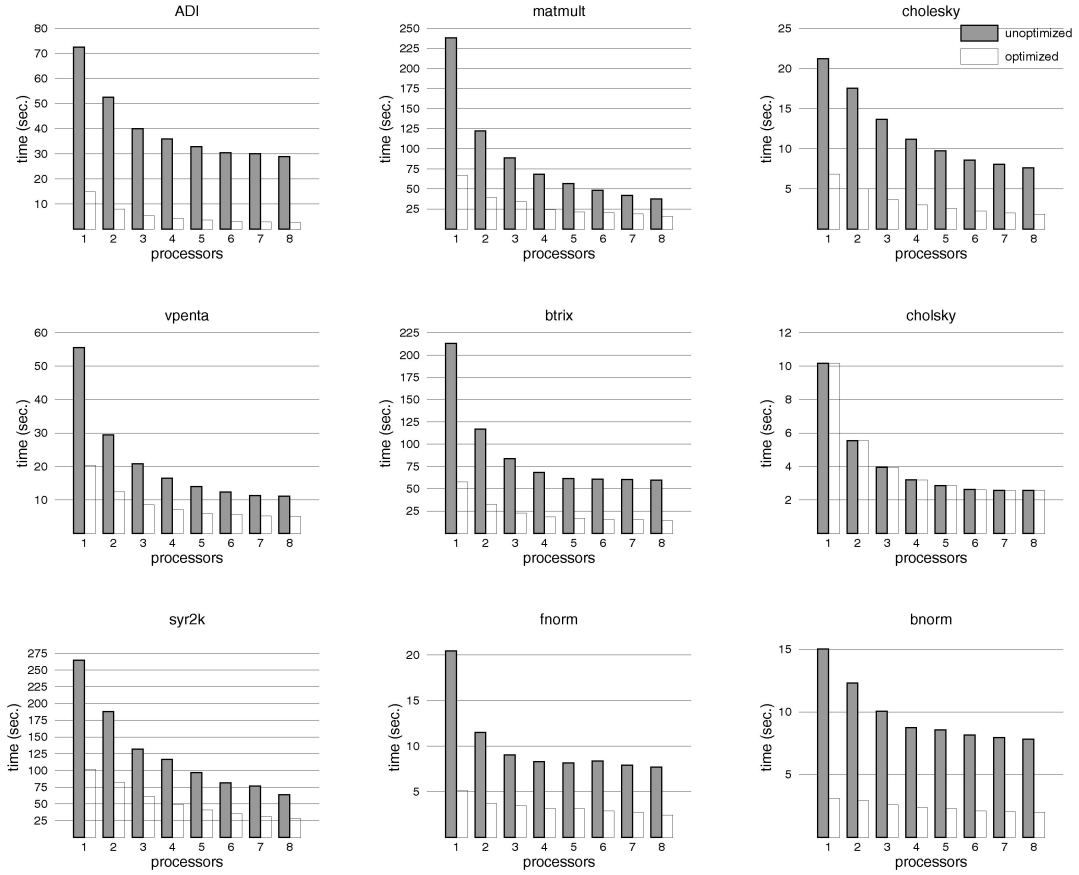


Fig. 5. Execution times on SFI/Cray Origin 2000 multiprocessors. The problem sizes are (in double precision elements) as follows: ADI—$1,000 \times 1,000 \times 3$ arrays; matmult—$1,200 \times 1,200$ matrices; cholesky—$1,024 \times 1,024$ matrices; vpenta—$4 \times 720 \times 720$ 3D arrays and 720×720 2D arrays; btrix—the size parameters are set to 150; cholsky—the size parameters are set to 2,500; syr2k—$1,024 \times 1,024$ matrices and $b = 400$; fnorm and bnorm—$6,144 \times 6,144$ matrices. The programs from Spec92Nasa7, AdI, and matmult have outermost timing loops that iterate twice. The timing loops enclose the entire program code except array initialization. The unoptimized version refers to the original program and the optimized version is the code generated by the technique discussed in this paper.

possible data decompositions to eliminate interprocessor communication entirely. During the optimization process, we first tried to optimize as many references as possible for temporal locality. In cases where we cannot achieve any temporal locality, we tried to achieve spatial locality for the references. In deciding the equation(s) to be eliminated (to reach a solution), we used reference weights (as explained earlier) obtained through profiling. In cases where we need to decide between two references of the same weight, we have chosen the first reference encountered during parsing.

Fig. 5 shows the performance results for the benchmark programs. For each program, the unoptimized version refers to the original program and the optimized version is the code generated by our technique. Both versions are then parallelized using the native compiler such that, for each loop, maximum degree of parallelism is obtained. We note that, except for cholsky, we have improvements for all programs against the unoptimized versions over all processor sizes. The cholsky code consists of a number of imperfectly nested loops; thus, it is difficult to optimize by a linear loop transformation technique such as ours. However, loop distribution [42] can substantially improve the performance by enabling linear loop transformations, as explained in the second part of our experimental results. In the case of syr2k, fnorm, and bnorm, the optimized programs do not scale well, mostly due to the use of static scheduling [42] for nonrectangular loop bounds. Apart from those, the results reveal that our approach is quite successful in optimizing locality on the SGI/Cray Origin 2000 machine. It should be noted that the data sizes that we use for the programs from Spec92/Nasa7 are higher than the sizes used by previous researchers as the Origin 2000 has a 4 MB L2 cache that could easily capture the original data set sizes, thereby obviating the need for locality optimizations.

We also studied the interaction between tiling and our optimization technique using the benchmarks in our suite. Fig. 6 shows the results (execution times in seconds on a single processor) for the *tiled versions* of the unoptimized (unopt) and optimized (opt) codes. The tile sizes used in the experiments are determined by the native compiler. The results show that applying linear loop transformation before tiling is very beneficial, indicating that linear locality optimizations and tiling are complementary. Also, in two codes (vpenta and btrix), the optimized untiled code performed better than the unoptimized tiled code.

In the second part, we evaluated the effectiveness of our approach in optimizing loop nests assuming that the memory layouts of the arrays might be *different*. Let us first focus on two programs: matmult and cholsky. Table 3a

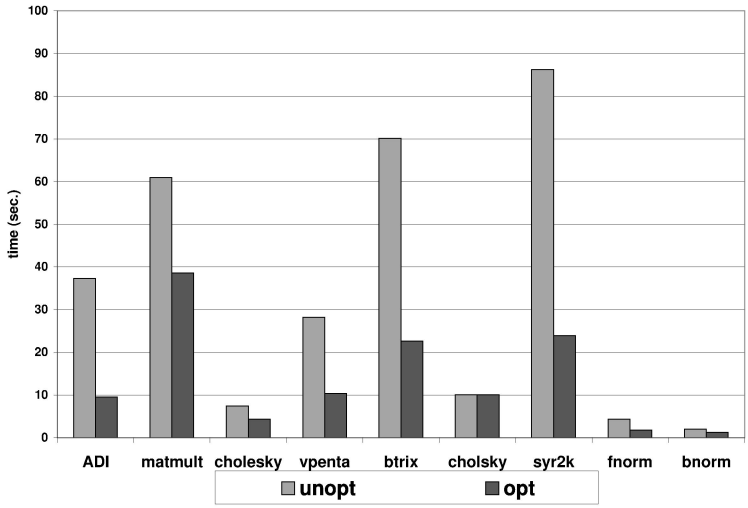


Fig. 6. Performance results for the *tiled versions* of unoptimized and optimized codes.

shows the single processor execution times for all permutation-based layout combinations of `matmult`, which computes $C = AB$. The legend `x-y-z` means that the memory layouts for C, A, and B are `x`, `y`, and `z`, respectively, where `c` means column-major and `r` means row-major (as in Table 1). For each combination, we experimented with all possible loop permutations. The boldfaced figures in each row denote the minimum times (in seconds) under the corresponding memory layout combinations. The preferred loop order detected for each layout combination by our algorithm is marked with a $\surd$. The loop order detected using only loop transformations that work with *fixed layouts*, namely, all row-major or all column-major for all arrays, is denoted by a †. When we compare these results with the sequential loop orders given in Table 1, it is easy to see that, except for two cases, our technique is able to find the optimal loop orders in every layout combination. In those two cases mentioned, our technique results in an

TABLE 3
Execution Times (in Seconds) of `matmult` under All Permutation-Based Memory Layouts and Loop Orders on SGI Origin

C-A-B	i-j-k	i-k-j	j-i-k	j-k-i	k-i-j	k-j-i
c-c-c	238.749	478.383	237.730	**66.921** † $\surd$	491.833	102.540
c-c-r	417.405	292.398	415.886	68.121	315.801	**68.062** $\surd$
c-r-c	64.798	490.661	**64.435** $\surd$	257.849	513.729	271.467
c-r-r	246.884	286.199	**237.370**	258.202 $\surd$	331.914	270.457
r-c-c	**238.227**	245.415 $\surd$	238.672	291.760	264.998	315.432
r-c-r	416.795	68.123	415.484	292.377	**67.992** $\surd$	315.216
r-r-c	**64.006** $\surd$	246.316	64.088	474.004	267.269	498.209
r-r-r	236.539	**66.932** † $\surd$	231.907	478.359	102.558	492.604

(a)

C-A-B	i-j-k	i-k-j	j-i-k	j-k-i	k-i-j	k-j-i
c-c-c	37.353	65.910	33.571	**16.803** $\surd$	116.271	58.403
c-c-r	56.988	43.322	55.972	**17.511** $\surd$	157.846	72.857
c-r-c	17.433	68.369	**15.390** $\surd$	38.722	261.163	76.908
c-r-r	39.549	42.305	**34.700**	39.406 $\surd$	342.488	73.710
r-c-c	35.547	34.831 $\surd$	**34.703**	42.235	73.034	130.526
r-c-r	56.227	**17.591** $\surd$	56.042	44.031	74.433	150.590
r-r-c	**15.347** $\surd$	34.907	18.502	64.313	74.986	106.485
r-r-r	32.968	**16.369** $\surd$	35.738	65.271	59.420	99.661

(b)

Minimum times for each layout combination are in boldface.

TABLE 4
Execution Times (in Seconds) of the Optimized and Unoptimized Versions of `cholsky` and Percent Improvements: (a) Number of Processors = 1, (b) Number of Processors = 8

	c-c		c-r		r-c		r-r	
	p=1	p=8	p=1	p=8	p=1	p=8	p=1	p=8
unoptimized	6.146	3.834	7.010	2.915	8.312	1.391	18.311	2.471
optimized	2.904	1.907	2.139	1.359	2.292	0.680	6.352	1.103
improvement	53%	50%	69%	53%	72%	51%	65%	55%

execution time which is close to the minimum time. Notice also that a loop transformation approach based on fixed memory layouts can only optimize two cases: `c-c-c` and `r-r-r`. Table 3b, on the other hand, shows the execution times in eight processors. Except for the two cases, the results are again consistent with those given in the last column of Table 1. Notice that the set of layouts used in Table 3a and Table 3b is not exhaustive; our approach can also optimize the `matmult` code when, say, array C is diagonal, array A is row-major, and array B is antidiagonal.

Next, we focus on the `cholsky` program from Spec92/Nasa7 benchmarks. This program accesses two three-dimensional arrays. First, we applied loop distribution to obtain as many perfectly nested loops as possible. Then, we conducted experiments with all four permutation-based layout combinations. The performance results given in Table 4 show that our technique is able to optimize the program for all layout combinations we experimented with. We note that the improvements are between 50 percent and 72 percent.

Finally, Fig. 7 summarizes the percentage improvements obtained applying our approach to the codes in our experimental suite. The numbers shown in this table are the mean values over all possible (permutation-based) layout combinations obtained using row-major and column-major layouts only. These results demonstrate that our approach is very successful in optimizing scientific codes when the arrays referenced have different memory layouts.

6 Related Work

Many earlier papers dealt with iteration space transformations. Here, we focus mostly on the related work on optimizing locality. McKinley et al. [30] presented a loop reordering technique to optimize locality and parallelism. In addition to loop permutations, their approach also employs loop distribution and loop fusion. Li [27], [28] and Wolf and Lam [39] developed frameworks which represent the data reuse information explicitly using the notion of reuse vectors. Wolf and Lam defined a *localized iteration space* and applied unimodular loop transformations to change the loops such that the vectors defining the localized space will capture the *reuse vector space*. In contrast, Li's approach takes into account loop bounds as well and can represent reuse information more accurately.

Our technique is different from those mentioned above in the sense that we can optimize a loop nest for locality, assuming that different arrays referenced in the nest may have distinct memory layouts, including row-major, column-major, higher dimensional equivalents of row- and column-major, as well as any type of skewed (e.g., diagonal) layout that can be expressed by hyperplanes. We believe that the approaches presented in [39] and [27], [28] can be extended to work with layouts consisting of a mix of column-major and row-major, but extending these approaches to skewed layouts (which might be very useful in banded matrix codes [26]) appears to be nontrivial. In fact, it is not clear to us how an iteration space-based locality enhancing technique can optimize a loop nest

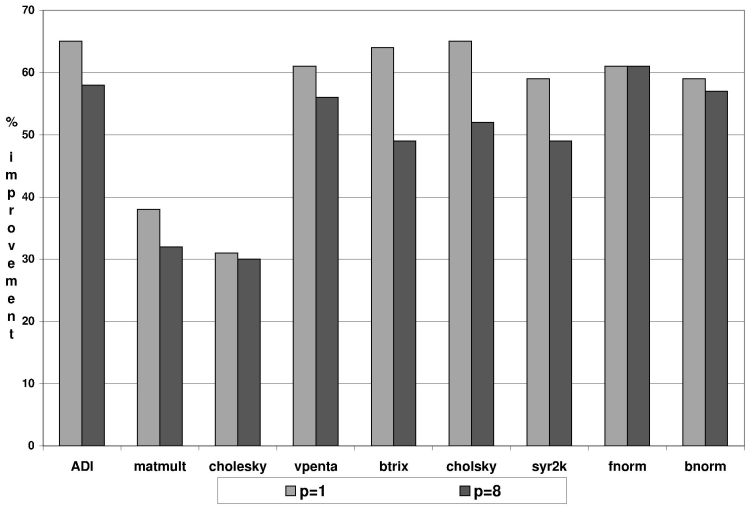


Fig. 7. Percentage improvements for single processor (`p = 1`) and multiprocessor (`p = 8`) cases.

enclosing arrays of different layouts (including skewed layouts) without using an explicit layout representation as we have done in this paper.

Tiling, a nonlinear locality optimization technique based on strip-mining and loop permutation, has been studied by Carr and Kennedy [6], Coleman and McKinley [8], Lam et al. [25], Ramanujam and Sadayappan [34], Wolf and Lam [39], and Wolfe [41]. It is well-known that tiling can improve cache locality for a given loop nest significantly by exploiting the reuse in outer loop nests [30]. A significant problem with tiling is the selection of good tile sizes. Previous studies have shown that the performance of tiling is very sensitive to the tile size and a wrong tile size, in fact, can degrade performance [39], [25], [8]. In that respect, we believe that our approach is a suitable step prior to the application of tiling. This is because improving spatial locality before tiling improves the intertile locality, thereby reducing the sensitivity of tiling to the tile size; this has been observed by Li [27], [28] as well. In addition, by increasing the number of loops, tiling increases the runtime overhead of nest execution. Our approach is useful in that respect, too, as we try to place all the spatial locality in the innermost loops and obviate the need for tiling the outermost loops that do not carry any type of reuse. Overall, our approach helps a compiler to apply tiling more judiciously. Our experiments confirm the benefits of using linear loop optimizations before tiling is applied.

More recently, there has been some work on optimizing locality using data layout transformations. The motivation is that some loop nests, either because they are imperfectly nested or because the data dependences enforce some execution order, cannot be optimized using loop transformations alone. In this context, Leung and Zahorjan [26] and O'Boyle and Knijnenburg [31] present array restructuring algorithms for optimizing locality. Although these techniques may be effective for some cases where locality enhancing loop transformations fail, the question of how to mitigate the global effect of a layout transformation remains to be solved. We believe that, for the global layout optimization problem, the loop transformation theory should be augmented so that the loop nests accessing a number of arrays with different layouts can be optimized. Leung and Zahorjan [26] and Kandemir et al. [15] handle multiple loop nests by enclosing them first with an imaginary outermost loop that iterates only once. The problem is that if there are conflicting references requiring different layouts in different loop nests, then the locality may not be exploited for all of them. Of course, it is possible to assign weights to references and consider these weights in selecting layouts. However, this scheme in general may lead to sacrificing locality for some references. The point here is that using loop transformations in conjunction with data transformations may enable the compiler to solve more layout conflicts without sacrificing too much locality. Another problem with the approaches based on pure data transformations is that they cannot optimize temporal locality.

Anderson et al. [2] propose a transformation technique that makes data elements accessed by the same processor contiguous in the shared address space. Their method is mainly for shared-memory parallel architectures. They use only permutations (of array dimensions) and strip-mining for possible data transformations. Our work can be extended to transform a given loop nest assuming that some of the arrays accessed have blocked memory layouts.

There have been a few attempts at a unified framework for locality optimizations. Cierniak and Li [7] and Kandemir et al. [18], [16], [17], [19] propose algorithms for optimizing cache locality using a blend of loop and data transformations. Unfortunately, since the solution space for combined loop and data transformations is very large, they make some simplifying assumptions. For example, Cierniak and Li [7] assume that the data transformations are restricted to be permutations; the loop transformation matrices can contain only ones and zeros. In [18], [19], on the other hand, only data transformations are restricted. In contrast, the technique presented in this paper can work with any type of memory layout that can be expressed by hyperplanes and can derive general nonsingular [35], [36] iteration space transformation matrices. We believe that our algorithm can be used as part of a global locality optimization framework that employs both loop and data transformations.

Kodukula et al. [24] have proposed a new form of tiling, called *data shackling*, using which imperfectly nested loops can also be handled successfully. Although it is ultimately a modification of the iteration space, data-shackling is a data-driven approach that can be successful in some cases where the traditional tiling may fail. Kodukula et al. [24] do not block the data in memory and whether or not such a data blocking will further enhance the performance is an open research issue, just as the precise interaction between linear layout transformations and iteration space tiling is. As with traditional tiling, we view this work as being complementary to ours.

Rivera and Tseng [37] have observed that conflict misses severely hinder the performance that would otherwise be obtained from exploiting spatial locality. They offer padding techniques that can improve the performance of sequential as well as parallel codes. Our work is orthogonal to their work in a sense. After a good locality is obtained using our approach, the arrays can be padded to prevent the devastating effect of potential conflict misses.

Transformations have been used for purposes other than improving locality. For example, Eggers and Jeremiassen [9], [13] use a number of data transformation techniques to eliminate false sharing on shared memory architectures and Ju and Dietz [14] use data transformations to reduce cache coherence overhead on shared-memory parallel machines. Wolf and Lam [40], Banerjee [4], Anderson et al. [2], and Ramanujam and Sadayappan [33], among others, use transformations for enhancing parallelism in loop nests.

There are a number of published techniques oriented toward *processor locality* (i.e., ensuring that an access by a processor can be satisfied locally without the need for interprocessor communication; this is important for distributed-memory machines) rather than the *cache locality* [29], [3], [10], [38], [22]. For example, on distributed-memory NUMA (nonuniform memory access) machines, such as SGI/Cray Origin 2000 or HP/Convex Exemplar,

careful distribution of data across processor memories may boost performance [38]. In order to obtain good processor locality, modern NUMA machines either use large cluster caches (as in HP/Convex Exemplar) or use dynamic page migration techniques (as in SGI/Cray Origin 2000) to mitigate the negative impact of an initial poor data distribution. An important question is whether having good processor locality obviates the need for techniques for improving cache locality and vice versa. Our experience shows that optimizing for cache locality influences processor locality significantly. The reason is that if we have good spatial locality in the innermost loop, a processor that starts to perform unit-stride accesses to the global data will cause the relevant parts of that data to transfer to its local memory (either through dynamic page migration or by taking a copy of it into the cluster cache). This means that, after a certain number of nonlocal accesses, most of the remaining accesses will be local. Our position is that optimizing cache locality is necessary for both uniprocessors and shared-memory NUMA architectures whether processor locality is essential or not.

7 Conclusions

In this paper, we have presented a technique to improve data locality in loop nests. Our technique uses explicit layout information available to our analysis in the form of layout constraint matrices. This information allows our technique to optimize loop nests in which each array may have a distinct memory layout. We believe that such a capability is necessary for global locality optimization techniques that optimize the loop nests in a program by applying an appropriate combination of loop and data transformations. This paper also discusses how our technique can be made to work with partial layout information as well. The experimental results validate the effectiveness of our approach in optimizing a given loop nest accessing arrays of different memory layouts. We see our work as a first step in integrating loop and data transformations fully.

We are currently looking at the interaction between our solution and tiling. Ongoing work includes extensively evaluating the relative performances of tiled code versus the resultant code from our approach and comparing our approach to data-centric tiling [24]. As mentioned earlier in the paper, our approach is most useful in an optimization framework that uses both loop and data transformations to optimize cache locality. Obviously, an important issue in a such a framework is the handling of procedure calls; we are working on this issue as well. In addition, our paper considers only linear memory layout transformations. It is worthwhile to investigate the effectiveness of blocked data layouts—in which the elements accessed by a tile are stored contiguously in memory—in conjunction with tiling in improving the cache performance further.

Acknowledgments

Mahmut Kandemir and Alok Choudhary were supported in part by US National Science Foundation (NSF) Young Investigator Award CCR-9357840 and NSF grant CCR-9509143. The work of J. Ramanujam is supported in part by NSF grant CCR-0073800 and by NSF Young Investigator Award CCR-9457768. Prithviraj Banerjee is supported in part by the NSF under grant CCR-9526325 and in part by the US Defense Advanced Research Projects Agency under contract F30602-98-2-0144. A preliminary version of this paper was presented at the Workshop on Languages and Compilers for Parallel Computing, Chapel Hill, North Carolina, August 1998.

References

[1] J. Anderson, "Automatic Computation and Data Decomposition for Multiprocessors," PhD dissertation, Stanford Univ., Mar. 1997. Also available as Technical Report CSL-TR-97-179, Computer Systems Laboratory, Stanford Univ.

[2] J. Anderson, S. Amarasinghe, and M. Lam, "Data and Computation Transformations for Multiprocessors," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* July 1995.

[3] E. Ayguade, J. Garcia, M. Girones, M.L. Grande, and J. Labarta, "A Research Tool for Automatic Data Distribution in HPF," *Scientific Programming,* vol. 6, no. 1, pp. 73-95, 1997.

[4] U. Banerjee, "Unimodular Transformations of Double Loops," *Proc. Advances in Languages and Compilers for Parallel Processing,* A. Nicolau et al., eds., MIT Press, 1991.

[5] A. Bik and H. Wijshoff, "On a Completion Method for Unimodular Matrices," Technical Report 94-14, Dept. of Computer Science, Leiden Univ., 1994.

[6] S. Carr and K. Kennedy, "Improving the Ratio of Memory Operations to Floating-Point Operations in Loops," *ACM Trans. Programming Languages and Systems,* vol. 16, no. 6, pp. 1768-1810, Nov. 1994.

[7] M. Cierniak and W. Li, "Unifying Data and Control Transformations for Distributed Shared Memory Machines," *Proc. SIGPLAN Conf. Programming Language Design and Implementation,* June 1995.

[8] S. Coleman and K. McKinley, "Tile Size Selection Using Cache Organization and Data Layout," *Proc. SIGPLAN Conf. Programming Language Design and Implementation,* June 1995.

[9] S. Eggers and T. Jeremiassen, "Eliminating False Sharing," *Proc. 1991 Int'l Conf. Parallel Processing (ICPP '91),* pp. 377-381, Aug. 1991.

[10] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Trans. Parallel and Distributed Systems,* vol. 3, no. 2, pp. 179-193, Mar. 1992.

[11] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1995.

[12] C.-H. Huang and P. Sadayappan, "Communication-Free Partitioning of Nested Loops," *J. Parallel and Distributed Computing,* vol. 19, pp. 90-102, 1993.

[13] T. Jeremiassen and S. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations," *Proc. SIGPLAN Symp. Principles and Practices of Parallel Programming,* pp. 179-188, July 1995.

[14] Y. Ju and H. Dietz, "Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation," *Languages and Compilers for Parallel Computing,* U. Banerjee et al., eds., pp. 344-358, Springer, 1992.

[15] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam, "A Linear Algebra Framework for Automatic Determination of Optimal Data Layouts," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 2, pp. 115-135, Feb. 1999.

[16] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "A Matrix-Based Approach to the Global Locality Optimization Problem," *Proc. Int'l Conf. Parallel Architecture and Compiler Techniques (PACT '98),* Oct. 1998.

[17] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "Improving Locality Using Loop and Data Transformations in an Integrated Approach," *Proc. MICRO-31,* Dec. 1998.

[18] M. Kandemir, J. Ramanujam, and A. Choudhary, "Compiler Algorithms for Optimizing Locality and Parallelism on Shared and Distributed Memory Machines," *Proc. Int'l Conf. Parallel*

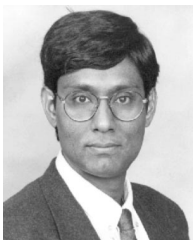
Architecture and Compiler Techniques (PACT '97), pp. 236-247, Nov. 1997.
[19] M. Kandemir, J. Ramanujam, and A. Choudhary, "Improving Cache Locality by a Combination of Loop and Data Transformations," *IEEE Trans. Computers,* vol. 48, no. 2, pp. 159-167, Feb. 1999. A preliminary version appears in *Proc. 11th ACM Int'l Conf. Supercomputing (ICS '97),* pp. 269-276, July 1997.
[20] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee, "A Locality Optimization Algorithm Based on Explicit Representation of Data Layouts," Technical Report CSE-00-008, Dept. of Computer Science and Eng., Pennsylvania State Univ., May 2000.
[21] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "The Omega Library Interface Guide," Technical Report CS-TR-3445, Computer Science Dept., Univ. of Maryland, College Park, Mar. 1995.
[22] K. Kennedy and U. Kremer, "Automatic Data Layout for High Performance Fortran," *Proc. Supercomputing '95,* Dec. 1995.
[23] I. Kodukula and K. Pingali, "Transformations of Imperfectly Nested Loops," *Proc. Supercomputing 96,* Nov. 1996.
[24] I. Kodukula, N. Ahmed, and K. Pingali, "Data-Centric Multi-Level Blocking," *Proc. Programming Language Design and Implementation (PLDI '97),* June 1997.
[25] M. Lam, E. Rothberg, and M. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '91),* 1991.
[26] S.-T. Leung and J. Zahorjan, "Optimizing Data Locality by Array Restructuring," Technical Report TR 95-09-01, Dept. of Computer Science and Eng., Univ. of Washington, Sept. 1995.
[27] W. Li, "Compiling for NUMA Parallel Machines," PhD thesis, Dept. of Computer Science, Cornell Univ., 1993.
[28] W. Li, "Compiler Cache Optimizations for Banded Matrix Problems," *Proc. Ninth ACM Int'l Conf. Supercomputing (ICS '95),* pp. 21-30, July 1995.
[29] J. Li and M. Chen, "Compiling Communication Efficient Programs for Massively Parallel Machines," *J. Parallel and Distributed Computers,* vol. 2, no. 3, pp. 361-376, 1991.
[30] K. McKinley, S. Carr, and C.W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Programming Languages and Systems,* vol. 18, no. 4, pp. 424-453, July 1996.
[31] M. O'Boyle and P. Knijnenburg, "Non-Singular Data Transformations: Definition, Validity, Applications," *Proc. Sixth Workshop Compilers for Parallel Computers (CPC '96),* pp. 287-297, 1996.
[32] M. O'Boyle and P. Knijnenburg, "Integrating Loop and Data Transformations for Global Optimisation," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '98),* Oct. 1998.
[33] J. Ramanujam and P. Sadayappan, "Compile-Time Techniques for Data Distribution in Distributed Memory Machines," *IEEE Trans. Parallel and Distributed Systems,* vol. 2, no. 4, pp. 472-482, Oct. 1991.
[34] J. Ramanujam and P. Sadayappan, "Tiling Multidimensional Iteration Spaces for Multicomputers," *J. Parallel and Distributed Computing,* vol. 16, no. 2, pp. 108-120, Oct. 1992.
[35] J. Ramanujam, "Non-Unimodular Transformations of Nested Loops," *Proc. Supercomputing '92,* pp. 214-223, Nov. 1992.
[36] J. Ramanujam, "Beyond Unimodular Transformations," *J. Supercomputing,* vol. 9, no. 4, pp. 365-389, 1995.
[37] G. Rivera and C.-W. Tseng, "Data Transformations for Eliminating Conflict Misses," *Proc. SIGPLAN Conf. Programming Language Design and Implementation,* June 1998.
[38] S. Tandri and T. Abdelrahman, "Automatic Partitioning of Data and Computations on Scalable Shared Memory Multiprocessors," *Proc. 1997 Int'l Conf. Parallel Processing (ICPP '97),* pp. 64-73, Aug. 1997.
[39] M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. SIGPLAN Conf. Programming Language Design and Implementation,* pp. 30-44, June 1991.
[40] M. Wolf and M. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. Parallel and Distributed Systems,* vol. 2, no. 4, Oct. 1991.
[41] M. Wolfe, "More Iteration Space Tiling," *Proc. Supercomputing '89,* pp. 655-664, Nov. 1989.
[42] M. Wolfe, *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.

Mahmut Kandemir received the BSc and MSc degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He received the PhD degree from Syracuse University, Syracuse, New York, in electrical engineering and computer science in 1999. He has been an assistant professor in the Computer Science and Engineering Department at Pennsylvania State University since August 1999. His main research interests are optimizing compilers, I/O intensive applications, and power-aware computing. He is a member of the IEEE and the ACM.

J. Ramanujam (Ram) received the BTech degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1983, and the MS and PhD degrees in computer science from Ohio State University, Columbus, Ohio, in 1987 and 1990, respectively. He is currently an associate professor of electrical and computer engineering at Louisiana State University, Baton Rouge, Louisiana. His research interests are in embedded systems, compilers for high-performance computer systems, software optimizations for low-power computing, high-level hardware synthesis, parallel architectures and algorithms. He has published more than 90 papers in refereed journals and conferences in these areas in addition to several book chapters. Dr. Ramanujam received the US National Science Foundation's Young Investigator Award in 1994. He has served on the program committees of several conferences and workshops, such as the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001), the Workshop on Power Management for Real-Time and Embedded Systems, IEEE Real-Time Applications Symposium, 2001, the International Conference on Parallel Architectures and Compilation Techniques (PACT 2000), the International Symposium on High Performance Computing (HiPC '99) and the 1997 International Conference on Parallel Processing. He is coorganizing a workshop on compilers and operating systems for low power to be held in conjunction with PACT 2001 in October 2001. He coorganized the first workshop in this series as part of PACT 2000 in October 2000. He has taught tutorials on compilers for high-performance computers at several conferences such as the International Conference on Parallel Processing (1998, 1996), Supercomputing '94, Scalable High-Performance Computing Conference (SHPCC '94), and the International Symposium on Computer Architecture (1993 and 1994). He has been a frequent reviewer for several journals and conferences. He is a member of the IEEE.

Alok Choudhary received the PhD degree from the University of Illinois, Urbana-Champaign, in electrical and computer engineering in 1989, the MS degree from the University of Massachusetts, Amherst, in 1986, and the BE (honors) degree from the Birla Institute of Technology and Science, Pilani, India, in 1982. He is a professor of electrical and computer engineering at Northwestern University. From 1993 to 1996, he was an associate professor in the Electrical and Computer Engineering Department at Syracuse University and, from 1989 to 1993, he was an assistant professor in the same department. He worked in industry for computer consultants prior to 1984. He received the US National Science Foundation's Young Investigator Award in 1993 (1993-1999). He has also received an IEEE Engineering Foundation award, an IBM Faculty Development award, and an Intel Research Council award. His main research interests are in high-performance computing and communication systems and their applications in many domains, including multimedia systems, information processing, and scientific computing. In particular, his interests lie in the design and evaluation of architectures and software systems (from system software such as runtime systems, compilers, and programming languages to applications), high-performance servers, high-performance databases, and input-output. He has published more than 130 papers in various journals and conferences in the above areas. He has also written a book and several book chapters on the above topics. His research has been sponsored by (past and present) DARPA, NSF, NASA, AFOSR, ONR, DOE, Intel, IBM, and TI.

Dr. Choudhary served as the conference cochair for the International Conference on Parallel Processing and as a program chair and general chair for the International Workshop on I/O Systems in Parallel and Distributed Systems. He also served as program vice-chair for HiPC 1999. He is an editor of the *Journal of Parallel and Distributed Computing* and an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has also served as a guest editor for *Computer* and *IEEE Parallel and Distributed Technology*. He serves (or has served) on the program committee of many International conferences in architecture, parallel computing, multimedia systems, performance evaluation, distributed computing, etc. He is a senior member of the IEEE and a member of the IEEE Computer Society and the ACM. He also serves on the High-Performance Fortran Forum, a forum of academic, industry, and government labs working on standardizing programming languages for portable programming on parallel computers.



Prithviraj Banerjee received the BTech degree in electronics and electrical engineering from the Indian Institute of Technology, Kharagpur, India, in August 1981, and the MS and PhD degrees in electrical engineering from the University of Illinois at Urbana-Champaign in December 1982 and December 1984, respectively. He is currently the Walter P. Murphy Professor and Chairman of the Department of Electrical and Computer Engineering and Director of the Center for Parallel and Distributed Computing at Northwestern University in Evanston, Illinois. Prior to that he was the director of the Computational Science and Engineering Program and professor of electrical and computer engineering and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. His research interests are in parallel algorithms for VLSI design automation, distributed memory parallel vompilers, and vompilers for adaptive computing and he is the author of more than 300 papers in these areas. He leads the PARADIGM compiler project for compiling programs for distributed memory multicomputers, the ProperCAD project for portable parallel VLSI CAD applications, the MATCH project on a MATLAB compilation environment for adaptive computing, and the PACT project on power aware compilation and architectural techniques. He is also the author of the book *Parallel Algorithms for VLSI CAD* (Prentice Hall, 1994). He has supervised 27 PhD and 30 MS student theses so far.

Dr. Banerjee has received numerous awards and honors during his career. He received the IEEE Taylor L. Booth Education Award from the IEEE Computer Society in 2001. He became a fellow of the ACM in 2000. He was the recipient of the 1996 Frederick Emmons Terman Award of ASEE's Electrical Engineering Division sponsored by Hewlett-Packard. He was elected to the fellow grade of IEEE in 1995. He received the University Scholar award from the University of Illinois in 1993, the Senior Xerox Research Award in 1992, the IEEE senior membership in 1990, the US National Science Foundation's Presidential Young Investigators' Award in 1987, the IBM Young Faculty Development Award in 1986, and the President of India Gold Medal from the Indian Institute of Technology, Kharagpur, in 1981. He served as the program chair of the High-Performance Computing Conference in 1999 and program chair of the International Conference on Parallel Processing for 1995. He served as general chairman of the International Conference on Parallel and Distributed Computing Systems in 1997 and the International Workshop on Hardware Fault Tolerance in Multiprocessors, 1989. He served on the program and organizing committees of the 1988, 1989, 1993, and 1996 Fault-Tolerant Computing Symposia, the 1992, 1994, 1995, 1996, and 1997 International Parallel Processing Symposia, the 1991, 1992, 1994, and 1998 International Symposia on Computer Architecture, the 1998 International Conference on Architectural Support of Programming Languages and Operating Systems, the 1990, 1993, 1994, 1995, 1996, 1997, and 1998 International Symposia on VLSI Design, the 1994, 1995, 1996, 1997, 1998, and 2000 International Conferences on Parallel Processing, and the 1995, 1996, and 1997 International Conferenced on High-Performance Computing. He is an associate editor of the *IEEE Transactions on Parallel and Distributed Systems* and *IEEE Transactions on Computers*. In the past, he served as an associate editor of the *Journal of Parallel and Distributed Computing*, the *IEEE Transactions on VLSI Systems*, and the *Journal of Circuits, Systems, and Computers*. He has been a consultant to many companies and was on the Technical Advisory Board of Ambit Design Systems.