A Loop Transformation Algorithm Based on Explicit Data Layout Representation for Optimizing Locality*

M. Kandemir[1], J. Ramanujam[2], A. Choudhary[1], and P. Banerjee[1]

[1] ECE Dept., Northwestern University,
Evanston, IL 60208, USA
{mtk,choudhar,banerjee}@ece.nwu.edu
[2] ECE Dept., Louisiana State University,
Baton Rouge, LA 70803, USA
jxr@ee.lsu.edu

Abstract. We present 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 finds a non-singular iteration-space transformation matrix such that in a given loop nest spatial locality is exploited in the innermost loops where it is most useful. The method builds inverse of a non-singular transformation matrix 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. Experimental results on an 8-processor SGI Origin 2000 show that our technique reduces execution times by up to 72%.

1 Introduction

As the disparity between processor and memory speeds increases, it is increasingly important to restructure programs so that the time spent in accessing or waiting for memory will be minimized. Previous research has shown that impressive speedups can be obtained if the programs are restructured to take advantage of the memory hierarchy by satisfying as many data references as possible from the cache instead of the main memory. Along these lines, several restructuring techniques have been offered including loop (iteration space) as well as array layout (data space) transformations [9,8,13]. The basic idea is to modify the access pattern of the program such that the data brought into cache is reused as much as possible before being discarded from the cache.

* M. Kandemir and A. Choudhary were supported by NSF Young Investigator Award CCR-9357840, NSF grant CCR-9509143 and Air Force contract F30602-97-C-0026. J. Ramanujam was supported by NSF Young Investigator Award CCR-9457768. P. Banerjee was supported by NSF grant CCR-9526325 and by DARPA contract DABT-63-97-C-0035.

S. Chatterjee (Ed.): LCPC'98, LNCS 1656, pp. 34–50, 1999.

```
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) Original program. (b) Transformed program.
The transformed program exhibits good spatial locality provided that array U has diagonal memory layout and array V is row-major.

Loop transformations Consider an array reference $U[i, j]$ in a two-deep loop nest where the outermost loop is i and the inner loop is j. Assuming that the array U is stored in memory in column-major order and that the trip counts N of the enclosing loops are very large, successive iterations of the j-loop will touch different columns of array U which will very likely get mapped onto different cache lines. Let us focus on a specific cache line that holds the initial part of a given 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. Consequently, in the worst case, every memory access to array U may involve a data transfer between the cache and memory resulting in high 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 on optimizing compilers [13,8,9,14] has proposed algorithms to detect and perform this loop interchange transformation automatically.

Data transformations Alternately, the same problem can be tackled by a technique called *data 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, the successive iterations of the j-loop can reuse the data in the cache. Recently, several authors [10,2,7,5] have proposed data transformation techniques. Although these are promising because they are not affected by data dependences, the effect of a layout transformation is global, meaning that it effects the cache behavior of all loop nests that access the array assuming a fixed layout for each array. In this paper, we consider the possibility of different arrays with different layouts. Therefore, in a general case, given a loop nest, the compiler is faced with finding an appropriate loop transformation assuming that

the arrays in the nest may have different—perhaps unspecified—layouts; current techniques are unable to handle this.

Combined data and loop transformations Consider the code fragment in Fig. 1(a), assuming that the arrays U and V are column-major by default. In this fragment, there are two disjoint loop nests with different access patterns. The first loop nest accesses array U diagonally and array V as row-major. The second loop nest accesses both arrays as row-major. Due to data dependence and conflicting access patterns of the arrays, the first loop nest cannot be optimized for locality of both arrays using the iteration space transformations alone. Using data layout transformations [4], we 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 loop nest. A simple 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 layout without any loop transformation. 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 layout changes, accesses to one of the arrays in the second loop nest remain unoptimized in either case. One suggestion may be to use loop skewing [14] in the second loop nest, but this results in poor locality for array V. What we need for this second loop nest is a loop transformation which 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 paper shows that such a transformation is possible. The resulting code is shown in Fig. 1(b). Notice that this program exhibits very good locality assuming that array U has diagonal layout and array V is row-major. Notice also that this optimized code requires the derived layouts for arrays U and V. The transformation that implements the desired layouts for arrays is rather mechanical; the associated details are beyond the scope of this paper and can be found elsewhere [7,4,10].

We present a framework which, given a loop nest, derives a transformation matrix for the case where the distinct arrays accessed in the nest may have different memory layouts. In addition, our solution works in the presence of undetermined layouts for a subset of the arrays referenced; this is of particular interest in enhancing locality beyond a single nest. The framework subsumes previous iteration space based linear locality enhancing transformation techniques which assume a fixed canonical memory layout for all arrays.

Outline After a brief review of the necessary terminology in Section 2, we summarize a framework to represent memory layout information mathematically in Section 3. In Section 4, we present a loop transformation framework assuming that the memory layout for all arrays is column-major. In Section 5, we generalize our approach to attack the problem of optimizing a loop nest assuming that the arrays referenced may have distinct memory layouts. Then in Section 6 we show how to utilize the partial layout information. We give our experimental

results obtained on an 8-processor SGI Origin 2000 distributed-shared-memory multiprocessor in Section 7. We review the related work on data locality in Section 8 and conclude the paper in Section 9.

2 Terminology

In this paper we consider nested loops. An iteration in an n-nested loop is denoted by the *iteration vector* $\bar{I} = [\imath_1, \imath_2, ..., \imath_n]^T$. 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 array U can be modeled by an access (or reference) matrix $\mathcal{L}$ of size $m \times n$ and an m-dimensional offset vector $\bar{o}$ [8,13,14]. 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 iteration space transformations that can be represented by integer non-singular square matrices. Such a transformation matrix is invertible and is of size $n \times n$ for an n-dimensional loop nest. 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 computation of new loop bounds is done using Fourier-Motzkin elimination [12]. An iteration space transformation is legal if it preserves all data dependences in the original loop nest [14]. A linear transformation represented by $\mathcal{T}$ is legal if, after the transformation, $\mathcal{T}\bar{d}$ is lexicographically non-negative for each dependence $\bar{d}$ in the original nest.

3 Memory layout representation using hyperplanes

In this section, we briefly review the concepts [4] relating to the representation of memory layouts using hyperplanes. A *hyperplane* defines a set of elements $(\jmath_1, \cdots, \jmath_m)$ that satisfies

$$g_1\jmath_1 + g_2\jmath_2 + \cdots + g_m\jmath_m = c \tag{1}$$

for a constant c. Here, $g_1, \cdots, g_m$ are rational numbers called hyperplane coefficients and c is a rational number called hyperplane constant [11]. 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 order of hyperplanes with respect to each other may not be important. The relative storage order of columns (although well defined by column-major

layout) is not important for the purposes of this paper. The hyperplane vector $(1, 0)$ denotes a row-major layout, $(0, 1)$ denotes column-major layout, and $(1, -1)$ defines a diagonal layout. Two array elements $\bar{J}$ and $\bar{J}'$ belong to the same hyperplane $\bar{g}$ if

$$\bar{g}^T \bar{J} = \bar{g}^T \bar{J}'. \tag{2}$$

As an example, in a two-dimensional array stored as column-major (hyperplane vector $[0, 1]$), array elements $[4, 5]$ and $[9, 5]$ belong to the same hyperplane (i.e., the same column) but elements $[4, 5]$ and $[4, 6]$ 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 to use more hyperplane families. Let us concentrate on 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 that case, two data elements $\bar{J}$ and $\bar{J}'$ have spatial locality if

$$\bar{g}^T \bar{J} = \bar{g}^T \bar{J}' \quad \text{and} \quad \bar{h}^T \bar{J} = \bar{h}^T \bar{J}'. \tag{3}$$

The elements that have spatial locality should be stored in consecutive memory locations. The idea can easily be generalized to higher dimensions [4]. In this paper, unless stated otherwise, we assume column-major memory layout for all arrays. In Section 5, we show how to generalize our technique to optimize a given loop nest where a number of arrays with distinct memory layouts are accessed. Our optimization technique can work with any memory layout which can be represented by hyperplanes. The layout matrices that we use for column-major storage (starting from two-dimensional case) are as follows: $\underbrace{[0, 1]}_{2D}, \underbrace{\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}}_{3D}, \underbrace{\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}}_{4D}, \cdots$. In general, the column-major layout for an m-dimensional array can be represented by an $(m-1) \times m$ matrix $L = [l_{ij}]$ such that $l_{i(m-i+1)} = 1$ for $1 \leq i \leq (m-1)$ and $l_{ij} = 0$ for the remaining elements. Notice that each row of the layout constraint matrix represents a constraint on the elements that have spatial locality with respect to the associated layout. In some cases, the order of the rows in a layout constraint matrix may be important. But, for the purposes of this paper, we assume they are not. Thus, any row permutation of the layout matrices mentioned before is also considered a legal layout constraint matrix.

```
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

  (a)
```

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

  (b)
```

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

  (c)
```

Fig. 2. Matrix multiplication code. (a) Original loop nest. (b)-(c) Transformed loop nests.

4 Transformation for optimizing spatial locality

Our objective 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 (i.e., reside on the same column by our definition of spatial locality). In particular, if possible, we want the accessed array elements to be next to each other so that they can be on 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 Equation (2) for two-dimensional arrays and the relation in Equation (3) for three-dimensional arrays and so on.

Assume $\mathcal{Q} = \mathcal{T}^{-1}$ for convenience. Ignoring the offset vector, after transformation $\mathcal{T}$, 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 $L_U = [0, 1]$. Two consecutive iteration vectors can be written as $\bar{I} = [\imath_1, \cdots, \imath_{n-1}, \imath_n]^T$ and $\bar{I}_{next} = [\imath_1, \cdots, \imath_{n-1}, 1 + \imath_n]^T$. The data elements accessed by $\bar{I}$ and $\bar{I}_{next}$ through a reference represented by access matrix $\mathcal{L}$ will have spatial locality (see equation (2)) if $[0, 1]\mathcal{L}\mathcal{Q}\bar{I} = [0, 1]\mathcal{L}\mathcal{Q}\bar{I}_{next}$ or $[0, 1]\mathcal{L}\mathcal{Q}[0, 0, \cdots, 0, 1]^T = 0$; i.e., $\bar{l_m}\bar{q_n} = 0 \quad \Rightarrow \bar{q_n} \in Ker\left\{\bar{l_m}\right\}$, where $\bar{l_m}$ and $\bar{q_n}$ are the last row of matrix $\mathcal{L}$ and last column of matrix $\mathcal{Q}$, respectively. Since $\bar{l_m}$ is known, we can always choose $\bar{q_n}$ from its null set (Ker set). Note that here m is 2.

Consider the matrix-multiplication loop nest shown in Fig. 2(a). The access matrices are $\mathcal{L}_C = \begin{bmatrix} 1\,0\,0 \\ 0\,1\,0 \end{bmatrix}$, $\mathcal{L}_A = \begin{bmatrix} 1\,0\,0 \\ 0\,0\,1 \end{bmatrix}$, and $\mathcal{L}_B = \begin{bmatrix} 0\,0\,1 \\ 0\,1\,0 \end{bmatrix}$.

$$
\begin{array}{lllll}
\text{For array } C: & \bar{q_3} \in Ker\{[0,1,0]\} & \Rightarrow & \bar{q_3} = [\times, 0, \times]^T \\
\text{For array } A: & \bar{q_3} \in Ker\{[0,0,1]\} & \Rightarrow & \bar{q_3} = [\times, \times, 0]^T \\
\text{For array } B: & \bar{q_3} \in Ker\{[0,1,0]\} & \Rightarrow & \bar{q_3} = [\times, 0, \times]^T.
\end{array}
$$

Thus $\mathcal{Q} = \begin{bmatrix} \times & \times & \times \\ \times & \times & 0 \\ \times & \times & 0 \end{bmatrix}$. Therefore, $\mathcal{Q}_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$ and $\mathcal{Q}_2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ are the only suitable permutation matrices. Fig. 2(b) and (c) give the transformed nests obtained using $\mathcal{T}_1 = Q_1^{-1}$ and $\mathcal{T}_2 = Q_2^{-1}$ respectively. Notice that although the spatial locality is good for both transformed versions, the one in Fig. 2(c) is expected to perform better (see [8]). Next, we will explain why and how the loop nest in Fig. 2(c) is preferred over the one in Fig. 2(b).

Notice that our approach as explained so far determines only (possibly part of) the last column of the matrix $\mathcal{Q}$. The remaining elements can be filled in any way as long as the resulting $\mathcal{Q}$ is non-singular and its inverse ($\mathcal{T}$) observes all data dependences. We will focus on how to complete a partially filled $\mathcal{Q}$ matrix later. For now, to see why the last column of $\mathcal{Q}$ is so important for locality, 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 $\mathcal{Q} = \left[\bar{q_1}\ \bar{q_2} \cdots \bar{q_n}\right]$, where $\bar{l_i}$ is the i^{th} row of $\mathcal{L}$ and $\bar{q_j}$ is the j^{th} column of $\mathcal{Q}$. Assuming i_1, i_2, ..., i_n are the loops in the nest *after* the transformation, omitting the offset vector, the new reference matrix is of the form $\mathcal{L}\mathcal{Q}[i_1, \cdots, i_n]^T = [\bar{l_1}\bar{q_1}i_1 + \cdots + \bar{l_1}\bar{q_n}i_n, \cdots, \bar{l_m}\bar{q_1}i_1 + \cdots + \bar{l_m}\bar{q_n}i_n]$. Since the spatial 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 determining the spatial locality. Our objective is to select $\bar{q_n}$ such that $\bar{l_j}\bar{q_n}i_n$ will be 0 for each $2 \leq j \leq n$, and $\bar{l_1}\bar{q_n}i_n$ will be (preferably) 1 or a small integer constant.[1] In two-dimensional case, since $m = 2$, selecting $\bar{q_n}$ from $Ker\left\{\bar{l_m}\right\}$ 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

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

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 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 consider the matrix-multiplication code of Fig. 2(a) again. In this example, one of the references exhibits temporal locality in the innermost loop. Consequently, it might be wise to take the second innermost loop into account as well. Recall that for this example previously we ended up with two possible permutation matrices. In fact, the partly filled $\mathcal{Q}$ matrix was $\mathcal{Q} = \begin{bmatrix} \times & \times & \times \\ \times & \times & 0 \\ \times & \times & 0 \end{bmatrix}$. Now, we focus on the spatial locality in the second innermost loop. This corresponds to determining the elements of the second rightmost column of $\mathcal{Q}$. Let us define

[1] In the case where $\bar{l_1}\bar{q_n}i_n$ is also 0, we have temporal locality in the innermost loop. We do not consider exploiting temporal locality explicitly in this paper. Our approach can be modified to take temporal locality into account as well.

$\bar{I}_k = [\imath_1, \cdots, \imath_{k-1}, \imath_k, \imath_{k+1}, \cdots, \imath_n]^T$ 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 k^{th} index where they differ by one. In this case, we say that $\bar{I}_k$ and $\bar{I}'_k$ are *consecutive in k^{th} loop*. Notice that previously we have considered only the consecutive iterations in the innermost loop. From our experience, we can say that for majority of the loop nests which appear in scientific codes, this is sufficient. For most of the remaining loop nests, it should be enough to consider the spatial locality in the *second* innermost loop. For two-dimensional column-major arrays, we can formalize the idea as follows: $[0,1]\mathcal{L}\mathcal{Q}\bar{I}_{n-1} = [0,1]\mathcal{L}\mathcal{Q}\bar{I}'_{n-1}$, or $[0,1]\mathcal{L}\mathcal{Q}[0,0,\cdots,0,1,0]^T = 0$; this implies that $\bar{l_m}\bar{q_{n-1}} = 0$, i.e., $\bar{q_{n-1}} \in Ker\left\{\bar{l_m}\right\}$. For the matrix-multiplication example, we proceed as follows:

$$\begin{array}{lllll} \text{For array } C: & \bar{q_2} \in Ker\{[0,1,0]\} & \Rightarrow & \bar{q_2} = [\times, 0, \times]^T \\ \text{For array } A: & \bar{q_2} \in Ker\{[0,0,1]\} & \Rightarrow & \bar{q_2} = [\times, \times, 0]^T \\ \text{For array } B: & \bar{q_2} \in Ker\{[0,1,0]\} & \Rightarrow & \bar{q_2} = [\times, 0, \times]^T. \end{array}$$

Combining these equations with those obtained on $\bar{q_3}$, we have to ignore one equation. Since the equations on C and B are the same, we favor that equation. Thus, we choose $\mathcal{Q}$ as $\begin{bmatrix} \times & \times & \times \\ \times & 0 & 0 \\ \times & \times & 0 \end{bmatrix}$. The only suitable permutation matrix is $\mathcal{Q} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$. The resulting code obtained using $\mathcal{T} = Q^{-1}$ is shown in Fig. 2(c). This code exploits spatial locality for arrays C and A in the innermost loop. Array B, on the other hand, has temporal locality in the innermost loop and spatial locality in the second innermost loop.

5 Algorithm to find the loop transformation for the general case

In this section, we present the formulation of the problem for the most general case where a number of arrays with possibly different memory layouts are referenced in a given loop nest. Our objective is to find a transformation matrix $\mathcal{T}$ such that the spatial locality will be good for as many references as possible. To resolve the conflicts between different references, we assume that prior to our analysis the references are ordered according to their importance. We use the following notation:

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

Without loss of generality, we also assume that the references are ordered as $R_1,...R_\nu$, where R_1 is the most important reference and R_ν is the least important. Ideally, the references should be ordered according to their access frequencies. Currently, we use profile information for this purpose.

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

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

Setting $b^\sigma_{ij} = \sum_{k=1}^{m} l^\sigma_{ik} a^\sigma_{kj}$ $(1 \le i \le m-1, 1 \le j \le n)$, we rewrite this relation 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. Notice that this solution takes care of the reference R_σ only. In order to obtain a transformation which satisfies all ν references we have to set up and solve the following system $B^1\bar{q_n} = \bar{0}, B^2\bar{q_n} = \bar{0}, \cdots, B^\nu\bar{q_n} = \bar{0}$. Additionally, as in matrix multiplication code, we might want to add the constraints on $\bar{q_{n-1}}$ to exploit the spatial reuse in the second innermost loop. Given a large number of references, this homogeneous system may not have a solution. In that case, we drop some equations from consideration starting from those of B^ν and repeat the process. The complete algorithm is given in Fig. 3 on page 43. A solution to this homogeneous system is of the form $\bar{q_n} = \delta_1\bar{x_1} + \delta_2\bar{x_2} + \cdots + \delta_p\bar{x_p}$. We fill out $\mathcal{Q}$ of the form

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

Notice that assuming $q_{nn} \neq 0$ such a transformation matrix is non-singular. Moreover, we can set $\delta_1, \cdots, \delta_p$ such that for each $\bar{d} \in \mathcal{D}$, $\mathcal{T}\bar{d} \ge 0$ where $\bar{q_n} = [q_{1n}, \cdots, q_{nn}]^T = \delta_1\bar{x_1} + \delta_2\bar{x_2} + \cdots + \delta_p\bar{x_p}$ and $\mathcal{D}$ is the original dependence matrix. Let $\bar{d} \in \mathcal{D}$ be a dependence vector as follows $\bar{d} = [d_1, \cdots, d_{n-1}, d_n]^T$. After the transformation $\mathcal{T}$, we have $\mathcal{T}\bar{d} = [d_1 - d_n q_{1n}/q_{nn}, d_2 - d_n q_{2n}/q_{nn}, \cdots, d_{n-1} - d_n q_{(n-1)n}/q_{nn}, d_n/q_{nn}]^T$. 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. Otherwise, the parameters $\delta_1, \cdots, \delta_p$ can be chosen so that $\mathcal{T}\bar{d}$ is lexicographically non-negative.

As an application of this algorithm, we consider the matrix-multiplication code given in Fig. 2(a) once more. This time we assume that arrays A and C

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 which 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 $B^\sigma = L_\sigma\mathcal{L}_\sigma$. Eliminate the redundant equations and let τ be the number of the remaining rows in B

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 [14].
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 last row of B and repeat **Step 2** until a $r < n$ is found.

Step 4. Write the solution $\bar{x}$ as a linear combination of vectors $\bar{x_1}, \bar{x_2}, ..., \bar{x_p}$ with the corresponding coefficients $\delta_1, \delta_2, ..., \delta_p$:

$$\bar{x} = \delta_1\bar{x_1} + \delta_2\bar{x_2} + ... + \delta_p\bar{x_p}$$

Step 5. Choose $\delta_1, \delta_2, \cdots, \delta_p$ such that Q is of the form in Equation 4, and for each $\bar{d} \in \mathcal{D}$, $\mathcal{T}\bar{d} \geq 0$ where $\bar{q_n} = [q_{1n}, q_{2n}, \cdots, q_{nn}]^T = \delta_1\bar{x_1} + \delta_2\bar{x_2} + ... + \delta_p\bar{x_p}$ and $\mathcal{D}$ is the original dependence matrix.

Step 6. Return $\mathcal{T} = \mathcal{Q}^{-1}$.

Fig. 3. Algorithm for determining the transformation matrix (it is assumed that $q_{nn} \neq 0$).

are row-major whereas array B is column-major. 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. The partially filled matrix is $Q = \begin{bmatrix} \times & \times & 0 \\ \times & \times & 0 \\ \times & \times & 1 \end{bmatrix}$, which can be completed as $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ and $T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

What this implies is that under the mentioned memory layouts, the original loop order `i-j-k` is the best loop order from the locality point of view. Table 1 shows the best loop orders for the matrix multiplication nest under all possible (permutation-based) layout combinations. The middle column gives the best order for the sequential execution whereas the rightmost column gives that for the parallel execution. Determining the suitability of a locality-optimized sequential program for a parallel architecture is definitely an important issue that needs to be visited in the future. It should be emphasized that in deriving these best orders we have considered the second column of Q as well.

Table 1. Best loop orders for different layout combinations in the matrix-multiplication code.
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.

C-A-B	best (seq.)	best (par.)
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

6 Utilizing partial layout information

A direct generalization of the approach presented in the previous section is optimizing a loop nest assuming some of the arrays have fixed but possibly different layouts whereas the remaining arrays have not been assigned memory layouts yet. In the following we summarize our strategy; the details can be found elsewhere [3]. We handle this problem in two steps:

(1) find a loop transformation which satisfies the references to the arrays whose layouts have already been determined, and
(2) taking into account this loop transformation, determine optimal layouts for the remaining arrays referenced in the nest.

The first step is handled as shown in the previous section. The second step is an array restructuring problem and is fully explained in [4]. To illustrate the process, consider the example shown in Fig. 4(a) on page 45. Assuming a two-dimensional array layout represented by hyperplane vector $[g_1, g_2]$ and a reference represented by access matrix $\mathcal{L}$, the spatial locality will be exploited if $[g_1, g_2]\mathcal{L}[q_{1n}, \cdots, q_{nn}]^T = 0$, where $[q_{1n}, q_{2n}, \cdots, q_{nn}]^T$ is the last column of the inverse of the loop transformation matrix. Since both $[g_1, g_2]$ and $[q_{1n}, q_{2n}, \cdots, q_{nn}]^T$ are unknown, this formulation is non-linear. However, if either of them is known, the other can easily be found using the relations:

$$[g_1, g_2] \in Ker\left\{\mathcal{L}[q_{1k}, \cdots, q_{kk}]^T\right\} \tag{5}$$

$$[q_{1k}, q_{2k}, \cdots, q_{kk}]^T \in Ker\left\{[g_1, g_2]\mathcal{L}\right\}. \tag{6}$$

Usually Ker sets may contain multiple vectors in which case we choose the one such that the gcd of its elements is minimum. Let us consider the example in Fig. 4(a), the access matrices for the first nest are as follows: $\mathcal{L}_U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$,

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

do i = 1, N
  do j = 1, N
    W[j,i] = V[i,j] * V[j,i]
  end do
end do
  (a)
```

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

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

Fig. 4. (a) Original loop nest. (b) Transformed loop nest.

and $\mathcal{L}_V = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. For the second nest: $\mathcal{L}_{V_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathcal{L}_{V_2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, and $\mathcal{L}_W = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Let us assume for the first nest we apply only data transformations using the technique in [4]; that is, $[q_{12}, q_{22}]^T = [0, 1]$ (Q is identity matrix). Using (5), for array U,

$$[g_1, g_2] \in Ker\left\{\mathcal{L}_U[0,1]^T\right\} \qquad \Longrightarrow \qquad [g_1, g_2] \in Ker\left\{[0,1]^T\right\}.$$

A solution is $[g_1, g_2] = [1, 0]$ meaning that array U should be row-major. For array V,

$$[g_1, g_2] \in Ker\left\{\mathcal{L}_V[0,1]^T\right\} \qquad \Longrightarrow \qquad [g_1, g_2] \in Ker\left\{[1,1]^T\right\}.$$

Selecting $[g_1, g_2] = [1, -1]$ results in diagonal layout for array V.

Having fixed the layouts for two arrays, we proceed with the second nest whose optimization is the topic of this section. Assuming again that $\mathcal{Q}$ is the inverse of the loop transformation matrix for this nest, using (6), we find the loop transformation which satisfies the both references to array V:

$$\begin{aligned} [q_{12}, q_{22}]^T \in Ker\{[1,-1]\mathcal{L}_{V_1}\} \qquad &\Longrightarrow \qquad [q_{12}, q_{22}]^T \in Ker\{[1,-1]\} \\ \text{and } [q_{12}, q_{22}]^T \in Ker\{[1,-1]\mathcal{L}_{V_2}\} \qquad &\Longrightarrow \qquad [q_{12}, q_{22}]^T \in Ker\{[-1,1]\} \end{aligned}$$

$[q_{12}, q_{22}]^T = [1, 1]^T$ satisfies both the equations, and for this, $Q = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. The process so far is exactly what we have done in the previous section. The next task is to determine the optimal memory layout for array W which is referenced only in the second nest. By taking into account the last column of Q and using (5) once more (now for array W), $[g_1, g_2] \in Ker\left\{\mathcal{L}_W[1,1]^T\right\} \Longrightarrow [g_1, g_2] \in Ker\left\{[1,1]^T\right\}$ which means that array W should have a diagonal layout. The transformed program is shown in Fig. 4(b).

7 Experimental Results

In this section, we illustrate how our iteration space transformation technique improves performance on an 8-processor SGI Origin 2000 distributed-shared-

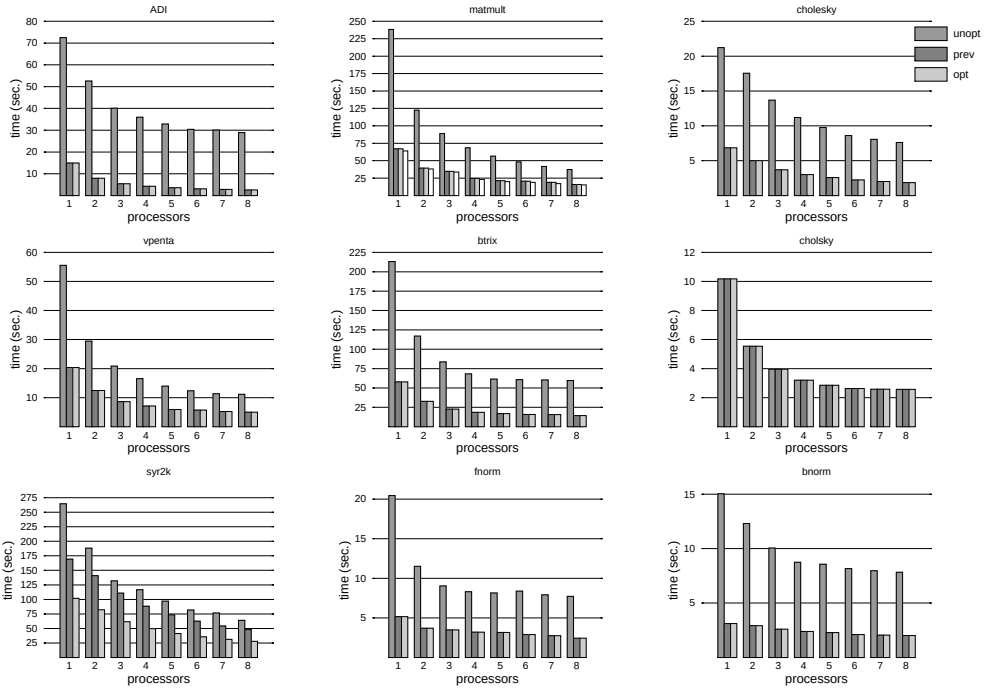

Fig. 5. Execution times on an SGI Origin. [The problem sizes are (in doubles) as follows. `ADI`: $1000 \times 1000 \times 3$ arrays; `matmult`: 1200×1200 arrays; `cholesky`: 1024×1024 arrays; `vpenta`: $4 \times 720 \times 720$ 3D arrays and 720×720 2D arrays; `btrix`: size parameter is 150; `cholsky`: size parameter is 2500; `syr2k`: 1024×1024 arrays with $b = 400$; `fnorm` and `bnorm`: 6144×6144 arrays. The programs from Spec92, the `ADI` code, and `matmult` have outer timing loops. The `unopt` version refers to the original program, the `prev` version refers to the approach offered by previous work, and the `opt` version is the code generated by our technique].

memory machine. This machine uses 195MHz R10000 processors, 32KB L1 data cache and 4MB L2 unified cache. The cache line size is 128 bytes and page size is 16KB. Our presentation is in two 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 focus on two programs, and measure the improvements in execution time with different layout combinations.

For the first part, we experiment with the following programs: `ADI` from Livermore kernels; `matmult`, a matrix-multiplication routine; `cholesky` from [8]; `vpenta`, `btrix`, and `cholsky`[2] from Spec92/NASA benchmark suite; `syr2k` from BLAS; and finally `fnorm` and `bnorm` from ODEPACK, a collection of solvers for the initial value problem for ordinary differential equation systems. We use the

[2] Different from `cholesky`; uses two three-dimensional arrays.

C versions of these programs and the hand–optimized programs are compiled by the native compiler using the `-O2` option (expect for `syr2k`).

Fig. 5 shows the performance results for our benchmark programs. For each program, the `unopt` version refers to the original program, the `prev` version refers to the approach offered by [8], and finally the `opt` version is the code generated by our technique. We note that except for `cholsky`, we have improvements in all programs against the unoptimized versions over all processor sizes. The `cholsky` code consists of a number of imperfectly nested loops; thus, is difficult to optimize by linear loop transformations. However, loop distribution [14] can substantially improve the performance by enabling linear loop transformations as explained in the second part of our experimental results. In `syr2k`, `fnorm` and `bnorm` the optimized programs do not scale well mostly due to employment of static scheduling for non-rectangular loop bounds. Apart from those, the results reveal that our approach is quite successful in optimizing locality in the Origin 2000. It should be noted that the sizes that we use for the programs from Spec92/NASA are larger than the usual sizes; so the results should not be compared with the previous works. It should also be noted that except for `syr2k` our approach and the approach offered in [8] result in the same programs. In the `syr2k` code the access pattern is quite complicated and we perform better than Li's approach with the `-O1` compiler option. If the `-O2` option is used, however, the two approaches perform very similar. In general, for a fixed permutation-based memory layout for all arrays, we expect that unless the access pattern is complicated both optimization approach will result in either the same or very similar codes. For the case where all the arrays have a uniform diagonal layout, it is not clear to us how Li's approach can be modified. Finally, the fourth bar in performance results of `matmult` shows the execution time of the best possible layout combination and the associated loop order. In this case, the best time has been obtained with the `i-j-k` loop order assuming that the arrays C and A have row-major memory layout whereas array B is column-major. We note that given this layout combination our approach can derive the `i-j-k` loop order automatically.

In the second part, we evaluate the effectiveness of our approach in optimizing loop nests assuming that the memory layouts of the arrays might be different. We focus on two programs: `matmult` and `cholsky`. Table 2(a) shows the single processor execution times for all permutation-based layout combinations of `matmult`. 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. For each combination, we experiment with all possible loop permutations. The boldfaced figures in each row denotes 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 best loop order using only loop transformations that work with fixed layouts namely all row-major or all column-major for all arrays (referred to as `prev`) is denoted by a †. When we compare these results with the best sequential loop orders given in Table 1, it is easy to see that, except for two cases, our technique is able to find

Table 2. Execution times (in sec.) of `matmult` under all permutation-based memory layouts and loop orders on SGI origin. [Minimum times for each layout combination are in boldface].

(a) Number of processors = 1

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** † √	491.833	102.540
c-c-r	417.405	292.398	415.886	68.121	315.801	**68.062**√
c-r-c	64.798	490.661	**64.435**√	257.849	513.729	271.467
c-r-r	246.884	286.199	**237.370**	258.202√	331.914	270.457
r-c-c	**238.227**	245.415√	238.672	291.760	264.998	315.432
r-c-r	416.795	68.123	415.484	292.377	**67.992**√	315.216
r-r-c	**64.006**√	246.316	64.088	474.004	267.269	498.209
r-r-r	236.539	**66.932** † √	231.907	478.359	102.558	492.604

(b) Number of processors = 8

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**√	116.271	58.403
c-c-r	56.988	43.322	55.972	**17.511**√	157.846	72.857
c-r-c	17.433	68.369	**15.390**√	38.722	261.163	76.908
c-r-r	39.549	42.305	**34.700**	39.406√	342.488	73.710
r-c-c	35.547	34.831√	**34.703**	42.235	73.034	130.526
r-c-r	56.227	**17.591**√	56.042	44.031	74.433	150.590
r-r-c	**15.347**√	34.907	18.502	64.313	74.986	106.485
r-r-r	32.968	**16.369**√	35.738	65.271	59.420	99.661

the optimal loop orders in every layout combination. In those two cases mentioned our technique results in an 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 2(b), 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. Since `prev` does not offer an optimal layout for the multiprocessor case, it is not shown.

Next, we focus on the `cholsky` program from Spec92 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 3 show that our technique is able to optimize the program for all layout combinations we experiment with. The improvements are between 51% and 72%.

Table 3. Execution times in seconds of the optimized and unoptimized versions of `cholsky` and % improvements [`p` is the number of processors].

	c-c		c-r		r-c		r-r	
version	p=1	p=8	p=1	p=8	p=1	p=8	p=1	p=8
unopt	6.146	3.834	7.010	2.915	8.312	1.391	18.311	2.471
opt	2.904	1.907	2.139	1.359	2.292	0.680	6.352	1.103
imprv.	53%	51%	69%	53%	72%	51%	65%	55%

8 Related work

Early work on automating locality enhancing optimizations was done by Abu-Sufah et al. [1]. More recently the interest was on loop restructuring for optimizing cache locality. McKinley et al. [9] present a loop reordering technique to optimize locality and parallelism. Their approach also employs loop distribution and loop fusion. Li [8] and Wolf and Lam [13] developed frameworks where the data reuse information is represented explicitly using reuse vectors. Our technique is different from those mentioned here in the sense that we can optimize a loop nest for locality assuming different arrays referenced in the nest may have distinct memory layouts which include row-major, column-major, higher dimension equivalents of row- and column-major as well as any type of skewed layout that can be expressed by hyperplanes. Extending the approaches presented in [13] and [8] to work with general layouts is non-trivial.

More recently, there is an interest in optimizing locality using data layout transformations. In this context, Leung and Zahorjan [7] and O'Boyle and Knijnenburg [10] 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. Leung and Zahorjan [7] and Kandemir et al. [4] propose solutions to handle multiple loop nests. A major problem with the approaches based on pure data transformations is that they cannot optimize temporal locality.

There have been a few attempts at a unified framework for locality optimizations. Cierniak and Li [2] and Kandemir et al. [5] propose algorithms for optimizing cache locality using a blend of loop and data transformations, albeit drawn from a restricted set. 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 non-singular [14] iteration space transformation matrices. Recently Kodukula et al. [6] have proposed *data shackling*, in which data is first blocked and based on the data blocks that are accessed together, iteration space tiling is derived. But, the data is stored in memory using the default layouts.

9 Conclusions

We have presented a technique to improve data locality in loop nests. Our technique uses explicit layout information available to our analysis as 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 needed by a global locality optimization algorithm which optimizes the loop nests in a program by applying an appropriate combination of loop and data transformations.

References

1. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Trans. Comp.,* C-30(5):341–356, 1981. 49
2. M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. *Proc. SIGPLAN Conf. Prog. Lang. Des. & Imp.,* June 1995. 35, 49
3. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A Matrix-Based Approach to the Global Locality Optimization Problem In *Proc. 1998 Int. Conf. Parallel Architectures & Compilation Techniques (PACT 98)*, October 1998. 44
4. M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A hyperplane based approach for optimizing spatial locality in loop nests. In *Proc. 12th ACM Int. Conf. Supercomputing*, July 1998. 36, 37, 38, 44, 45, 49
5. M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proc. 11th ACM Int. Conf. Supercomputing*, pp. 269–276, July 1997. 35, 49
6. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Prog. Lang. Des. & Imp.,* June 1997. 49
7. S.-T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR 95-09-01, CSE Dept., University of Washington, Sep. 1995. 35, 36, 49
8. W. Li. Compiling for NUMA parallel machines. Ph.D. Thesis, Cornell University, 1993. 34, 35, 37, 40, 46, 47, 49
9. K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems,* 1996. 34, 35, 49
10. M. O'Boyle and P. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Par. Comp.,* pp. 287–297, Germany, 1996. 35, 36, 49
11. 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. 37
12. A. Schrijver. *Theory of linear and integer programming*, John Wiley, 1986. 37
13. M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pp. 30–44, June 1991. 34, 35, 37, 49
14. M. Wolfe. *High performance compilers for parallel computing*, Addison Wesley, 1996. 35, 36, 37, 43, 47, 49