Improving locality using loop and data transformations in an integrated framework

M. Kandemir* A. Choudhary* J. Ramanujam† P. Banerjee*

Abstract

This paper presents a new integrated compiler framework for improving the cache performance of scientific applications. In addition to applying loop transformations, the method includes data layout optimizations, i.e., those that change the memory layouts of data structures (arrays in this case). A key characteristic of this approach is that loop transformations are used to improve temporal locality while data layout optimizations are used to improve spatial locality. This optimization framework was used with sixteen loop nests from several benchmarks and math libraries, and the performance was measured using a cache simulator in addition to using a single node of the SGI Origin 2000 distributed-shared-memory machine for measuring actual execution times. The results demonstrate that this approach is very effective in improving locality and outperforms current solutions that use either loop or data transformations alone. We expect that our solution will also enable better register usage due to increased temporal locality in the innermost loop, and that it will help in eliminating false-sharing on multiprocessors due to exploiting spatial locality in the innermost loop.

1 Introduction

High performance computers of today extensively use multiple levels of memory hierarchies. This renders the performance of applications critically dependent on their memory access characteristics. In particular, careful choice of memory-sensitive data layouts and code restructuring appear to be crucial. Unfortunately, the lack of automatic tools forces many users and in particular library writers to manually restructure their code. The problem is exacerbated by the increasingly sophisticated nature of applications. Manual restructuring requires a clear understanding of the impact of the machine architecture, is tedious and error-prone, and results in severely reduced portability. In this paper we present and evaluate a compiler framework for improving the cache performance of scientific applications using a careful combination of loop transformations and data layout optimizations. The kind of data layout optimizations considered here include memory layout changes such as row-major or column-major storage of multi-dimensional arrays (which are common data structures in regular scientific applications). We will refer to data layout optimizations as *data transformations.*

Traditionally, loop transformations [4, 8, 14, 17, 21] have been the main techniques used to improve locality by changing the access pattern as a result of changing the order of execution of loop iterations. The effect of loop transformations is local, i.e., a loop transformation affects only the loop nest to which it is applied, and both temporal and spatial locality may improve as a result. But loop transformations are not always legal, and they affect all arrays in a loop nest some of them perhaps adversely. In a sense, loop transformations impact locality *indirectly* as a result of changing execution order. In addition, loop transformations are not readily applicable to imperfectly nested loops.

Data transformations [1, 16, 19] such as changing the memory layout of (multi-dimensional) arrays from column-major to row-major or vice-versa are almost always legal; however constructs such as pointer arithmetic in C and common blocks in Fortran may prevent memory layout transformations by exposing unmodifiable layouts to the compiler. Data transformations are equally useful for imperfectly nested loops as well. The effect of a data transformation is global in the sense that decisions regarding the memory layout of an array influence the locality characteristics of every part of the program that accesses the said array. A key drawback is that data transformations do not improve temporal locality [16].

Given these advantages and drawbacks of loop and data transformations, neither of these by itself is fully effective in optimizing locality in scientific codes. Cierniak and Li [5] show that some programs benefit most from a combined approach that consists of iteration space (loop) as well as data space (array layout) transformations. In deriving a combined approach, the following important questions need to be addressed:

(1) How will these two transformation techniques be integrated? In what order will these be applied to programs?

(2) What is the set of transformations considered (allowed) in loop and data transformations?

The first question is difficult to answer in general. Our approach is based on the following simple observation mentioned earlier: data transformations may only affect spatial locality whereas loop transformations affect temporal as well as spatial locality. Setting off with this observation, we take the following approach: first using loop transformations, optimize as much temporal locality as possible; then for references that do not exhibit temporal locality use data transformations to improve spatial locality. Therefore, a distinctive feature of our approach is that it improves locality by using (a) loop transformations exclusively to improve temporal locality; and (b) using data transformations exclusively to improve spatial locality. Our solution starts with the detection of the amount of different types of reuse [21] in a loop nest. Then a subset of the references is optimized for temporal locality using loop transformations. Next, the references within the loop nest are divided into two groups: the ones with optimized temporal locality, and the ones that do not exhibit temporal locality. For the latter group of references, we apply data transformations to enhance their spatial locality. It should be emphasized that since data transformations never degrade temporal locality, this order of application seems quite reasonable. Regarding the second question posed above, in this paper, we use unimodular loop (e.g., interchange, reversal, etc.) and unimodular data transformations (e.g., row-major to column-major); this is due to the fact that unimodular transformations preserve the volume of the iteration or data space [2].

This paper is organized as follows. In the next section we define data reuse and discuss the basics of loop and data transformations. In Section 3 we present our integrated approach through several examples. In Section 4 we show how our approach can be extended to

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

†Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. e-mail: jxr@ee.lsu.edu

optimize multiple loop nests simultaneously. We report preliminary results in Section 5. Section 6 reviews related work and Section 7 concludes with summary and an outline of planned future work.

2 Preliminaries

In this section we discuss loop and data transformation techniques used in our integrated approach. Consider an access to an m-dimensional array in a loop nest of depth n. We assume that the array subscript functions and loop bounds are affine functions of enclosing loop indices and symbolic variables. Let $\bar{I}$ denote the *iteration vector* (consisting of loop indices starting from the outermost loop); each array reference can be represented by $\mathcal{L}\bar{I} + \bar{o}$, where the $m \times n$ matrix $\mathcal{L}$ is called the access (or reference) matrix [21] and the m-element vector $\bar{o}$ is called an offset vector. Note that each row of $\mathcal{L}$ corresponds to a dimension of the array; and each column of $\mathcal{L}$ gives information about the effect of the corresponding loop on array accesses. In particular, the locality behavior of the innermost loop is determined by the last column of $\mathcal{L}$.

2.1 Temporal reuse and spatial reuse

We mainly focus on self-temporal and self-spatial reuses; however, our approach can be extended to handle group reuses [21] as well. When a reference in a loop nest accesses the same data in different iterations we say that *temporal reuse* occurs. Similarly, if a reference accesses data residing on the same cache line in different iterations we say that *spatial reuse* occurs. It should be emphasized that the most important reuses (whether temporal or spatial) are the ones exhibited by the innermost loop. If the innermost loop exhibits temporal reuse for a reference, then the element accessed by that reference can be kept in a register throughout the execution of the innermost loop. Similarly, spatial reuse is most beneficial when it occurs in the innermost loop, since in that case it may enable unit-stride accesses.

Consider the loop nest given in Figure 1(d). In this loop nest, assuming column-major memory layouts, arrays U and V have temporal (spatial) reuses in the inner (outer) and outer (inner) loops, respectively. Array W, on the other hand, does not exhibit temporal reuse, but has spatial reuse in the i-loop. It is reasonable to assume that for sufficiently large arrays and large loop trip counts (number of iterations), only the reuses associated with the innermost loop can be exploited, i.e., can be *converted* into *locality*. For this example, these reuses are the temporal reuse for array U and spatial reuse for array V. That is, if we do not apply any transformation, we can keep *U(i)* in a register and expect unit-stride accesses for array V. Notice that the locality behavior of this loop nest can be somewhat improved by interchanging the loops i and j. The data reuse theory introduced by Wolf and Lam [21] is used to identify the types of reuses in an automatic manner. Considering a reference $\mathcal{L}\bar{I}+\bar{o}$, two iteration vectors $\bar{I}_1$ and $\bar{I}_2$ access the same data element through this reference if $\mathcal{L}\bar{I}_1 + \bar{o} = \mathcal{L}\bar{I}_2 + \bar{o}$. In that case, the *temporal reuse vector* is defined as $\bar{r} = \bar{I}_2 - \bar{I}_1$, and can be found from $\mathcal{L}\bar{r} = \bar{0}$. In the loop nest shown in Figure 1(d) the access matrices are $\mathcal{L}_U = (1, 0)$, $\mathcal{L}_V = (0, 1)$ and $\mathcal{L}_W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. This implies that[1] $\bar{r}_U = (0, 1)^T$ and $\bar{r}_V = (1, 0)^T$. Therefore, the *temporal reuse spaces* are $R_U = span\{(0, 1)^T\}$, $R_V = span\{(1, 0)^T\}$ and $R_W = \emptyset$. Therefore, the temporal reuses for array U and array V are carried by the j-loop and i-loop respectively; and there is no temporal reuse for array W.

Assuming column-major memory layouts, spatial reuse occurs if the accesses are made to the same column. We can find the *spatial reuse vector* $\bar{s}$ from $\mathcal{L}_s\bar{s} = \bar{0}$, where $\mathcal{L}_s$ is $\mathcal{L}$ with all elements of the first row replaced by zero. In our example, $\bar{s}_U = \{(1, 0)^T, (0, 1)^T\}$, $\bar{s}_V = \{(1, 0)^T, (0, 1)^T\}$, and $\bar{s}_W = (1, 0)^T$. Consequently, the *spatial reuse spaces* are $S_W = span\{(1, 0)^T\}$, and $S_U = S_V = span\{(1, 0)^T, (0, 1)^T\}$. That is, both loops carry spatial reuse for arrays U and V; but only the outer loop carries spatial reuse for array W.

Notice that for this example we have $R_U \subset S_U$, $R_V \subset S_V$, and $R_W \subset S_W$. As a matter of fact, for any reference to an array U, $R_U \subset S_U$ holds in general as temporal reuse is a special case of spatial reuse [21].

2.2 Impact of a loop transformation

Let a loop transformation be represented by a square non-singular integer matrix T. Assuming that $\bar{I}$ is the original iteration vector and $\bar{I}' = T\bar{I}$ is the new iteration vector, each occurrence of $\bar{I}$ in the loop body is replaced by $T^{-1}\bar{I}'$. In other words, each reference represented by $\mathcal{L}\bar{I} + \bar{o}$ is transformed to $\mathcal{L}T^{-1}\bar{I}' + \bar{o}$. Loop transformations for locality are relatively well-studied; we will not describe any of the known approaches in detail, but refer the reader to [4], [7], [8], [17] and [21] for in-depth discussion of several approaches.

Consider the loop nest shown in Figure 1(a). Assuming that the default memory layout for array U is column-major (as in Fortran), the locality exhibited by this reference is poor. The reason is that consecutive iterations of the innermost loop touch array elements that are far apart in memory. Since these elements are separated by almost an entire column, they most probably would map onto different cache lines increasing the contention in cache memory. A locality-enhancing framework such as the one proposed by Li [17] or Wolf and Lam [21] can automatically detect this situation and can use a unimodular transformation $T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. The transformed loop nest is shown in Figure 1(b). In the transformed nest, consecutive iterations of the inner loop v access the consecutive elements of the same column, exhibiting high spatial reuse. It should be emphasized that since—in this example—$\mathcal{L}$ is of rank 2 (i.e., full rank), no non-singular (full rank) transformation matrix T can make all entries of the last column of $\mathcal{L}T^{-1}$ zero. In other words, it is impossible to transform this loop nest such that the reference shown in the figure will exhibit temporal locality in the innermost loop.

2.3 Impact of a data transformation

It is interesting to note that the nest shown in Figure 1(a) can be optimized using data transformation as well. Conceptually, a data transformation is applied by transforming the dimensions (subscript expressions) of the reference. Assuming again that we represent subscript function for the reference as $\mathcal{L}\bar{I}+\bar{o}$, a square non-singular data transformation matrix M transforms this reference to $M\mathcal{L}\bar{I} + M\bar{o}$. Notice that in constrast to loop transformations, the iteration vector does not change but offset vector is transformed. For this example, if we use $M = \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix}$, we obtain the loop nest shown in Figure 1(c). Notice again that assuming a default column-major layout, the spatial locality of this nest is very good.

Our approach to data transformations is based on the hyperplane concept from linear algebra which is briefly reviewed next. Here we focus mainly on two-dimensional arrays; however, the results easily extend to arrays of higher dimensions [12].

[1] We will write a column vector as $(x_1, x_2, \cdots, x_n)^T$ in this paper.

```
do i = 1, N
 do j = 1, M
  ··· U(i+j,j) ···
 end do
end do
```
(a)

```
do u = 1, M
 do v = 1, N
  ··· U(u+v,u) ···
 end do
end do
```
(b)

```
do i = 1, N
 do j = 1, M
  ··· U(i+j,i) ···
 end do
end do
```
(c)

```
do i = 1, N
 do j = 1, N
  U(i) = V(j) + W(i,j)
 end do
end do
```
(d)

Figure 1: (a) Original program. (b) Loop optimized program. (c) Data optimized program. (d) A loop nest that exhibits different reuses.

2.4 Hyperplane-based layout representation

Assuming a two-dimensional array, a *hyperplane* defines a set of array elements (a, b) that satisfy

$$g_1 a + g_2 b = c \tag{1}$$

for a constant c. In this equation, g_1 and g_2 are rational numbers called hyperplane coefficients and c is a rational number called hyperplane constant [10]. The hyperplane coefficients in equation (1) can be written as a hyperplane vector $g = (g_1, g_2)$. A *hyperplane family* is a set of hyperplanes with the same coefficients but with a different values of the constant.

A hyperplane family can be used to partially represent the memory layout of an array. For example, in a two-dimensional data space a hyperplane family defines a number of parallel hyperplanes (lines) each corresponding to a different value of c. We assume that the array elements on a specific hyperplane are stored in consecutive memory locations. Given a large array, the relative storage order of hyperplanes with respect to each other may not be important. As an example, for an array whose memory layout is column-major, each column represents a hyperplane whose elements are stored in consecutive locations in memory. The relative storage order of columns (although well defined in the case of column-major layouts used in Fortran) is not important for the purposes of this paper. Therefore, we can represent the column-major layout with the hyperplane vector $g = (0, 1)$ that simply indicates the orientation of the hyperplanes (lines). Similarly, the vectors $(1, 0)$, $(1, -1)$, and $(1, 1)$ correspond to row-major, diagonal and anti-diagonal memory layouts, respectively.

Two data (array) elements $\bar{J} = (a, b)$ and $\bar{J}' = (a', b')$ belong to the same hyperplane $\bar{g} = (g_1, g_2)$ if and only if

$$(g_1, g_2)(a, b)^T = (g_1, g_2)(a', b')^T. \tag{2}$$

Consider an array stored in column-major order, i.e., the layout hyperplane vector is $(0, 1)$. Here, the array elements $(4, 7)$ and $(10, 7)$ belong to the same hyperplane (i.e., same column) whereas the elements $(4, 7)$ and $(4, 8)$ do not. We say that two array elements that 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 very suitable for our locality optimization strategy.

The following result gives us a simple method to find a suitable memory layout for a given reference to have spatial locality in the innermost loop.

Result 1 *Consider a reference $\mathcal{L}\bar{I} + \bar{o}$ to a two-dimensional array in an n-depth loop nest where $\mathcal{L} = \begin{pmatrix} l_{11} & \cdots & l_{1n} \\ l_{21} & \cdots & l_{2n} \end{pmatrix}$. To have spatial locality in the innermost loop, the layout of this array must be represented by the hyperplane (g_1, g_2) where $(g_1, g_2) \in Ker(l_{1n}, l_{2n})^T$.*

Returning to our example in Figure 1(a), using this result, we have $(g_1, g_2) \in Ker\{(1, 1)^T\}$. This means that $(g_1, g_2) = (1, -1)$ is the spanning vector for that Ker set (null set). From the preceding discussion, we see that this vector corresponds to a diagonal memory layout where the elements along a diagonal of the matrix are stored in consecutive memory locations. Therefore, in order to have spatial locality in the innermost loop, array U should have a *diagonal* memory layout.

After detecting the suitable memory layout, the next step is to *implement* this layout taking into account the default layout adopted by the language in question. This process involves finding a suitable data transformation matrix M such that the desired locality improvement will be achieved. Assuming g_{def} is the hyperplane vector representing the default layout (e.g., column-major in Fortran), a desired memory layout g_{opt} for an array can be implemented in three steps summarized as follows: [12]:

(1) from $g_{def} M = g_{opt}$, a suitable data transformation matrix M is found;

(2) the subscript expression for each reference to the array is transformed: $\mathcal{L}\bar{I} + \bar{o} \mapsto M\left(\mathcal{L}\bar{I} + \bar{o}\right)$; (and)

(3) the array bounds are also transformed accordingly.

For Figure 1(a), since $g_{def} = (0, 1)$ and $g_{opt} = (1, -1)$, a suitable M is $\begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix}$, leading to the code in Figure 1(c).

It should be noted that the fact that the optimal layout should be diagonal will *only* be used by the compiler, and the programmer will not be aware of the data transformation performed. In fact, after the transformation, the array—with the renamed array elements—is stored in column-major as usual.

3 Integrated approach

In this section we present our integrated approach to enhance cache locality in a single nest. As mentioned earlier, our approach is based on optimizing temporal locality using loop transformations and optimizing spatial locality via data transformations.

Consider the loop nest given in Figure 2(a) which accesses three arrays. One-dimensional arrays *LHS* and *RHS* can be optimized for temporal locality. Assuming that the default layout is column-major, a technique based on linear loop transformations will most probably not do anything about this nest. The reason is that three out of the four references exhibit unit-stride accesses (spatial reuse) and the fourth reference exhibits temporal reuse in the innermost loop. That is, overall the locality is very good as it is. Likewise, a data layout transformation framework will not do anything either. The reason is that a data transformation framework in general does not transform one-dimensional arrays, and given the loop nest a column-major layout for the reference to the two-dimensional array *RMATRX* is the most suitable layout form. Our framework, on the other hand, first optimizes the temporal locality in the innermost loop for the most number of references. For this example, it achieves this by interchanging the loops, which leads to temporal locality for the first two references. It then focuses on the other references and exploits the spatial reuse by converting the memory

```
do i = 1, MATDIM
 do j = 1, MATDIM
  LHS(j) = RHS(j) - RMATRX(j,i) * RHS(i)
 end do
end do
```
(a)

```
do u = 1, MATDIM
 do v = 1, MATDIM
  LHS(u) = RHS(u) - RMATRX(v,u) * RHS(v)
 end do
end do
```
(b)

Figure 2: (a) A loop nest that can benefit from combined approach. (b) Optimized version of (a).

layout of the two-dimensional array from column-major to row-major and applying the corresponding transformation to the access matrix. The transformed program is shown in Figure 2(b) after the data access matrix has been transformed. Now it is easy to see that for two references we have temporal reuse in the innermost loop, and for the other two references we have spatial reuse in the innermost loop. Given the fact that as far as the innermost loop is concerned obtaining temporal reuse is more important (and better) than obtaining just spatial reuse, we have an improvement over the original code given in Figure 2(a).

3.1 Determining both loop and data transformation matrices: a non-linear problem

We now show that for a single reference, determining both a loop and a data transformation matrices simultaneously is equivalent to solving a non-linear system with some additional constraints. Suppose that the original reference is $\mathcal{L}\bar{I} + \bar{o}$, and we would like to apply a loop transformation matrix T and a data transformation matrix M. Then the transformed reference is $M\mathcal{L}T^{-1}\bar{I}' + M\bar{o}$. Omitting the offset vector part, since both M and T^{-1} are unknown, determining a suitable $M\mathcal{L}T^{-1}$ from the locality point of view involves solving a non-linear problem, with the additional constraints such that both M and T should be non-singular and T should observe all the data dependences in the original loop nest.

Suppose g_{opt} is the optimal layout that we need, q_{last} is the last column of the inverse of the loop transformation matrix T, and $\mathcal{L}$ is the access matrix for the reference in question. The following result gives us an important relation between g_{opt}, $\mathcal{L}$ and q_{last}.

Result 2 *In order to have spatial locality in the innermost loop, after the loop "and" data transformations, the following relation should hold:*

$$g_{\rm opt}\mathcal{L}q_{\rm last} \quad = \quad 0. \tag{3}$$

Therefore, the problem is reduced to finding a g_{opt} and a q_{last} for a given $\mathcal{L}$ such that the relation given by (3) will be satisfied. Since both g_{opt} and q_{last} are unknown, this formulation is still non-linear. However, if either of them is known, the other can easily be determined. For example, assuming that we know q_{last}, then

$$g_{opt} \quad \in \quad Ker\{\mathcal{L}q_{last}\}. \tag{4}$$

Likewise, if we know g_{opt}, then

$$q_{last} \quad \in \quad Ker\{g_{opt}\mathcal{L}\}. \tag{5}$$

On the other hand, suppose that we would like to optimize a reference with access matrix $\mathcal{L}$ for temporal locality. Since the resulting access matrix $\mathcal{L}' = \mathcal{L}T^{-1}$ should have a zero last column, the following relation should be satisfied:

$$q_{last} \quad \in \quad Ker\{\mathcal{L}\}, \tag{6}$$

where q_{last} again is the last column of inverse of the loop transformation matrix. If we cannot find a q_{last} that satisfies (6), i.e., if $Ker\{\mathcal{L}\}$ is empty, then this reference cannot be optimized for temporal locality. It should be noted that in this case applying a data transformation will not change a thing, since data transformations do not affect temporal locality.

3.2 Our solution

Instead of trying to work on the non-linear system given by (3) directly, our solution involves first applying a loop transformation and after that applying data transformations. We first explain the approach using the example nest given in Figure 3(a) which references two different arrays. The access matrices are

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

First let us attempt to optimize the reference to array U for temporal locality. Using (6), $q_{last} \in Ker\{\mathcal{L}_U\}$; or,

$$q_{last} \in Ker\left\{\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}\right\} \implies q_{last} = (0, 1, -1)^T.$$

Next we optimize the reference to array V for spatial locality using this q_{last}. Using (4), $g_{opt} \in Ker\{\mathcal{L}_V q_{last}\}$; or,

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

On the other hand, from q_{last}, we can complete T^{-1} to the unimodular matrix $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix}$. Having obtained the loop transformation matrix (actually its inverse) and the data transformation matrix M_V for array V, we next find the transformed access matrices:

$$M_U\mathcal{L}_UT^{-1} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}; M_V\mathcal{L}_VT^{-1} = \begin{pmatrix} 0 & 1 & -1 \\ 1 & 1 & 0 \end{pmatrix}$$

Note that since array U is optimized only for temporal locality, its data transformation matrix is the 2×2 identity matrix (i.e., $M_U = \mathcal{I}_{2\times 2}$). The transformed nest is shown in Figure 3(b). Also note that array U is optimized for temporal locality whereas array V optimized for spatial locality, assuming column-major memory layout. For the sake of simplicity, we do not show how the array declarations are modified.

In the following we explain how our approach works in the general case. Let us suppose that a loop nest of depth n references k arrays, and each array has only a single uniformly generated reference (UGR) set [8]. In that case, we can represent each array with a single reference matrix, as offset vectors are irrelevant from the locality analysis point of view (note that we do not consider group reuse). Therefore, in the following we use the terms "array" and "reference" interchangeably.

Let the access matrices of references in the loop nest be $\mathcal{L}_1$, $\cdots$, $\mathcal{L}_k$. Our approach first computes the spanning vectors for the

```
do i = 1, N
 do j = 1, N
  do k = 1, N
   U(j+k,i)=V(i+j+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(v-w,u+v)
  end do
 end do
end do
```

(b)

```
do i = 1, N
 do j = 1, N
  do k = 1, N
   U(i+j,j+k)=V(i+k,k)+W(k)
  end do
 end do
end do
```

(c)

```
do u = 1, N
 do v = 1, N
  do w = 1, N
   U(u+w,u-v)=V(u+v,v)+W(v)
  end do
 end do
end do
```

(d)

Figure 3: (a) Original loop nest. (b) Optimized version of (a). (c) Original loop nest. (d) Optimized version of (c).

kernel sets of these access matrices. Let $\bar{r}_{ij}$ be the jth ($1 \leq j \leq h_i$) spanning vector for $Ker\{\mathcal{L}_i\}$ ($1 \leq i \leq k$) where h_i is the number of spanning vectors for $Ker\{\mathcal{L}_i\}$. Considering all references, from among all spanning vectors, we choose the one which occurs most frequently. This heuristic tends to *maximize* the number of references for which temporal locality can be exploited. The selected spanning vector becomes the last column q_{last} of the inverse of the loop transformation matrix. Now, without lost of generality, suppose that for the selected q_{last} we are able to exploit temporal locality for the references with access matrices $\mathcal{L}_1, \cdots, \mathcal{L}_f$ whereas for the references with access matrices $\mathcal{L}_{f+1}, \cdots, \mathcal{L}_k$ no temporal locality is exploited.

Next we use relation (4), and from $g_{opt_j} \in Ker\{\mathcal{L}_j q_{last}\}$ we determine an optimal layout (g_{opt_j}) for each reference represented by $\mathcal{L}_j$ where $(f+1) \leq j \leq k$.

At this point we have our references such that f of the total k references exhibit temporal locality in the innermost loop and $k - f$ of the references exhibit spatial locality again in the innermost loop. Provided that the array sizes and loop trip counts are sufficiently large we can proceed as follows. First we complete the inverse of the loop transformation matrix such that all the data dependences are observed. Since we know q_{last}, the first row t_{first} of the loop transformation matrix T can be found from $t_{first} \in Ker\{q_{last}\}$ as q_{last} and t_{first} should be orthogonal. Having obtained the first row, the remaining rows of the loop transformation matrix and consequently the remaining columns of the inverse of it can be determined using the approaches given by Li [17] or Bik and Wijshoff [3] such that T will be unimodular and observe all data dependences in the loop nest. Then, we focus on determining suitable data transformation matrices. For $1 \leq i \leq f$, the data transformation matrices M_i are set to identity matrices ($\mathcal{I}$) of proper dimensionality as these references are optimized for temporal locality only. For the references $(f+1) \leq j \leq k$, we use $g_{def} M_j = g_{opt_j}$ to find suitable data transformation matrices M_j. Finally, using $M_i \mathcal{L}_i T^{-1} \bar{I}' + M_i \bar{o}$ we find the new access matrices and offset vectors for all references ($1 \leq i \leq k$).

In some cases, for a given reference represented by $\mathcal{L}_i$, ($1 \leq i \leq f$) we might want to exploit spatial locality as well in addition to temporal locality (assuming that it spatial locality is not already exploited). Since the innermost loop is the one that carries temporal reuse for a reference $\mathcal{L}_i$ ($1 \leq i \leq f$) (which has been optimized for temporal locality), spatial locality can be exploited only in the outer loops for this reference. In most cases, it is sufficient to focus only on the second innermost loop. Let q_{next} be the second column (from the right) of the inverse of the loop transformation matrix. Note that this column determines the effect of the second innermost loop on locality. Also, note that this column is the spanning vector with the second highest frequency. A reference represented by $\mathcal{L}_i$ which enjoys temporal locality in the innermost loop can exhibit spatial locality in the second innermost loop if it has a memory layout represented by g_{opt_i} such that $g_{opt_i} \in Ker\{\mathcal{L}_i q_{next}\}$, where $1 \leq i \leq k$.

The entire algorithm is given in Figure 4. If the references to the same array span more than one UGR set, then a *conflict resolution scheme* as discussed in [12] can be used.

Input $(\mathcal{L}_1, \bar{o}_1), \cdots, (\mathcal{L}_k, \bar{o}_k)$
Output$(\mathcal{L}'_1, \bar{o}'_1), \cdots, (\mathcal{L}'_k, \bar{o}'_k)$

foreach $1 \leq i \leq k$
  using $span\{r_{ij}\} = Ker\{\mathcal{L}_i\}$ compute all r_{ij} ($1 \leq j \leq h_i$)
endforeach
among r_{ij}, select $q_{last} = r^*_{ij}$ to exploit most temporal reuse
let $\mathcal{L}_1, \cdots, \mathcal{L}_f$ be references with temporal reuse
let $\mathcal{L}_{f+1}, \cdots, \mathcal{L}_k$ be references without temporal reuse
foreach $1 \leq i \leq f$
  $M_i = \mathcal{I}$
endforeach
foreach $(f+1) \leq j \leq k$
  determine g_{opt_j} from $g_{opt_j} \mathcal{L}_j q_{last} = 0$
  determine M_j from $g_{def} M_j = g_{opt_j}$
endforeach
complete q_{last} to T^{-1} using a completion algorithm
foreach $1 \leq i \leq k$
  let $\mathcal{L}'_i = M_i \mathcal{L}_i T^{-1}$
  let $o'_i = M_i o_i$
endforeach

Figure 4: Algorithm for determining optimized access matrices and offset vectors.

4 Multiple loop nests

In this section we discuss how we extend our approach to handle the multiple loop nest case. Before that however, we focus on the following sub-problem: *given a loop nest where layouts of some of the arrays referenced in it are fixed, what are the optimal loop transformation as well as the optimal memory layouts for the arrays whose layouts are not fixed yet?* As before, let us assume that there are k references with access matrices $\mathcal{L}_1, \cdots, \mathcal{L}_k$. Suppose that for e arrays the memory layouts are fixed. Also assume that of the remaining $k - e$ references, f references exhibit temporal reuse and $(k - e) - f$ references do not exhibit temporal reuse. In the following α is used to denote a reference from $\mathcal{L}_1, \cdots, \mathcal{L}_e$; and β denotes a reference from $\mathcal{L}_{e+1}, \cdots, \mathcal{L}_k$. Further, γ is used to denote a reference from $\mathcal{L}_{e+1}, \cdots, \mathcal{L}_{e+f}$ and θ denotes a reference from $\mathcal{L}_{e+f+1}, \cdots, \mathcal{L}_k$. Thus $\mathcal{L}_\alpha$, $\mathcal{L}_\gamma$, $\mathcal{L}_\theta$ denote references with fixed layout, with temporal locality, and with no temporal locality, respectively whereas $\mathcal{L}_\beta$ denotes to a reference with no fixed layout.

Our approach first tries to determine the last column q_{last} of the inverse of the loop transformation matrix T. Now, *different* from the case discussed in the previous section, two factors may affect our decision on what this last column should be:

(1) references whose associated layouts are fixed; and

(2) references which exhibit temporal reuse.

The effect of the first group of references is taken into account by the relation given in (5), i.e., $q_{last} \in Ker\{g_{opt_\alpha}\mathcal{L}_\alpha\}$. Each reference whose associated layout is fixed implies a desired q_{last} as the last column. The effect of the second group is taken into account as before using relation (6), i.e., $q_{last} \in Ker\{\mathcal{L}_\gamma\}$. Considering these two types of relations, the compiler chooses a q_{last} which satisfies the most number of references. In case of a tie, the preference is given to the alternative that exploits the temporal reuse for the most number of references. Having fixed the last column of the inverse of the loop transformation matrix, the next task is to determine the layout for each $\mathcal{L}_\theta$. We use relation (4), i.e., $g_{opt_\theta} \in Ker\{\mathcal{L}_\theta q_{last}\}$, in order to determine the suitable layouts. The overall algorithm is given in Figure 5. Notice that so far we have assumed that a reference whose associated layout is fixed does not exhibit temporal reuse in the loop nest considered. Obviously this assumption is not valid in general. If this reference exhibits temporal reuse, then we take this reference twice into account for determining the last column of the inverse of the loop transformation matrix.

Consider the loop nest shown in Figure 3(c), assuming that the memory layout of V is *fixed* as $g_{opt_V} = (1, -1)$. The access matrices are $\mathcal{L}_U = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$, $\mathcal{L}_V = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$, and $\mathcal{L}_W = (0, 0, 1)$. Considering the layout of array V, using the condition $\bar{r}_V \in Ker\{g_{opt_V}\mathcal{L}_V\}$, we have $\bar{r}_V \in Ker(1, 0, 0)$. We find $\bar{r}_V = \{(0, 1, 0)^T, (0, 0, 1)^T\}$. Next we look at the available temporal reuse. The spanning vectors for the access matrices are $\bar{r}_W = \{(1, 0, 0)^T, (0, 1, 0)^T\}$, $\bar{r}_U = (1, -1, 1)^T$, and $\bar{r}_V = (0, 1, 0)^T$. Given these vectors, we select $q_{last} = (0, 1, 0)^T$. This vector satisfies the spatial locality requirement of array V and helps us exploit temporal locality for V and W. Now the only reference which does not get optimized for temporal locality is the reference to array U. Using $g_{opt_U} \in Ker\{\mathcal{L}_U q_{last}\}$, we have $g_{opt_U} \in Ker(1, 1)^T$; so we can set $g_{opt_U} = (1, -1)$ which in turn gives us $M_U = \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix}$. We set M_V, and M_W to identity matrices. On the other hand, from q_{last}, we can obtain T^{-1} as $T^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. Having obtained the loop and data transformation matrices, we can compute the transformed access matrices as follows. $M_U\mathcal{L}_U T^{-1} = \begin{pmatrix} 1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix}$, $M_V\mathcal{L}_V T^{-1} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$, and $M_W\mathcal{L}_W T^{-1} = (0, 1, 0)$. The transformed code is shown in Figure 3(d). We note that for arrays V, and W temporal reuse is exploited whereas for array U spatial reuse is exploited, all in the innermost loop.

Notice that if there are no layout constraints (i.e., no layout is fixed), the algorithm given in Figure 5 reduces to the one given in Figure 4.

Now we return to the problem of optimizing a number of loop nests using loop and data transformations. Our approach to this problem is based on careful applications of the algorithms given in Figures 4 and 5. For simplicity, we assume that the loop nests we want to optimize are consecutive in the program and there is no conditional statement between or within them. Our approach is rather simple: First we determine an order of processing the nests; that is, if a nest is more important (costly) than another one, we optimize the more important nest first. Profiling can be used to determine the estimated cost of a loop nest. Then, for the most important nest we use the algorithm given in Figure 4. After optimizing this nest, it is possible that the memory layouts of some of the arrays referenced will be fixed. Then, we consider the next important nest and optimize it using the algorithm given in Figure 5, taking the layouts found in the most important nest into account. Then we move to the third most important loop nest, and in optimizing it (using the algorithm shown in Figure 5) we take all the layouts determined so far into account, and so on. Given the fact that in most scientific programs, the bulk of the execution time is spent in a couple of loop nests, our approach seems reasonable.

Input $(\mathcal{L}_1, \bar{o}_1), \cdots, (\mathcal{L}_k, \bar{o}_k)$ and g_{opt_α} for $1 \le \alpha \le e$
Output$(\mathcal{L}'_1, \bar{o}'_1), \cdots, (\mathcal{L}'_k, \bar{o}'_k)$

```
let L_1, ..., L_e be the references with fixed layouts
let L_{e+1}, ..., L_k be the references with no fixed layout
foreach 1 ≤ α ≤ e
  using r̄_αj ∈ Ker{g_opt_α L_α} compute all r_αj
endforeach
foreach (e+1) ≤ β ≤ k
  using span{r_βj'} = Ker{L_β} compute all r_βj'
endforeach
considering all r_αj and r_βj', select a suitable q_last
      which satisfies most of the references
let L_{e+1}, ..., L_f be references with temporal reuse
let L_{f+1}, ..., L_k be references without temporal reuse
foreach (e+1) ≤ γ ≤ f
  M_γ = I
endforeach
foreach (f+1) ≤ θ ≤ k
  determine g_opt_θ from g_opt_θ L_θ q_last = 0
  determine M_θ from g_def M_θ = g_opt_θ
endforeach
complete q_last to T^{-1} using a completion algorithm
foreach 1 ≤ α ≤ e
  let L'_α = L_α T^{-1}
  let o'_α = o_α
endforeach
foreach (e+1) ≤ β ≤ k
  let L'_β = M_β L_β T^{-1}
  let o'_β = M_β o_β
endforeach
```

Figure 5: Algorithm for determining optimized access matrices and offset vectors when some of the arrays referenced have fixed layouts.

5 Experimental results

We now present the results of our experiments. We first introduce our experimental codes, and different versions of them. We then study (using simulations) uniprocessor miss rates to show the performance improvement achieved by adopting our integrated approach. Then we present our execution time results obtained on a single node of SGI Origin 2000 distributed shared memory machine with two-levels of cache. We conclude the section with a summary of the results.

5.1 Loop nests

We measured the effectiveness of our approach on the performance of sixteen loop nests from several benchmarks and math libraries. The relevant information about these loop nests are given in Table 1. The `TO REF` column gives the total number of references in the loop nest whereas the `TP REF` gives the number of references with temporal reuse. For example the loop nest shown in Figure 2(a) has four references, three of which exhibits temporal reuse. Depending on the loop order, we can convert either one or

Table 1: Loop nests used in our experiments. The `TO REF` column gives the total number of references in the loop nest whereas the `TP REF` gives the number of references with temporal reuse. An entry *name.i* under the `CODE` column denotes the i^{th} loop nest (from top) of the benchmark *name*.

#	CODE	SOURCE	TO REF	TP REF	#	CODE	SOURCE	TO REF	TP REF
1	hydro2d/T1.1	Spec92	5	1	9	fnorm.1	Odepack	2	1
2	hydro2d/fct.7	Spec92	14	8	10	adi.2	Livermore	33	12
3	vpenta.6	Spec92	18	0	11	mxm.2	Spec92	10	10
4	emit.4	Spec92	4	3	12	rhf_hessian.1	Nwchem [18]	12	6
5	btrix.4	Spec92	12	5	13	transpose.1	Nwchem [18]	4	0
6	bakvec.2	Eispack	3	1	14	gfunp.4	Hompack	6	1
7	htribk.2	Eispack	8	8	15	r1mpyq.1	Minpack	6	3
8	qzhes.7	Eispack	2	2	16	rhf_hessian.2	Nwchem [18]	12	4

Table 2: Different versions of the codes used in our experiments.

VERSION	BRIEF DESCRIPTION
CM	original code: fixed column-major memory layout for all arrays
RM	original code: fixed row-major memory layouts for all arrays
LP	loop-optimized version: no memory layout transformation
DT	layout-optimized version: no loop transformation
UN	our approach : integrated loop & data layout transformations

two of these temporal reuses into temporal locality in the innermost loop. As can be seen from Table 1 we tried to select loop nests with different (`TP REF`/`TO REF`) ratios to observe the locality behavior of loop nests with different *degrees* of temporal reuse.

For each loop nest, we experimented with five different versions summarized in Table 2. For the `LP` version we used the technique given by Li [17] whereas for the `DT` version we used an approach of ours [12] that uses only data transformations to optimize spatial locality (without directly exploiting temporal locality). We believe that those represent the most current work on optimizing locality using *pure loop* and *pure data* transformations, respectively. The `UN` version is the version that is obtained by applying our integrated technique explained in the paper.

5.2 Simulation results

Figures 6 and 7 show the overall miss rates achieved by different versions of the loop nests. The miss rates are obtained using Dinero [9] for a direct-mapped cache with 64 bytes block (line) size. The horizontal axis shows different cache sizes whereas the vertical axis is the absolute miss rates. Except for some hard-coded values of array dimensions, the size of the dimension of any array used in the experiments is set to 2048 double-precision elements; an exception to this is the `adi.2` code where the three-dimensional arrays are of size $3 \times 512 \times 512$. Note that in all cases, to avoid bad strides due to power of two dimension sizes, array dimensions were padded by a small constant as needed. The trace sizes are limited to $100,000,000$ references. Our first observation is that, as expected, increasing the cache size generally reduces the miss rates. Secondly, beyond a certain cache size, the performances of different versions become quite similar; this is to be expected, because in the cache can hold the entire working set of the loop nest up to a certain size. Another observation is that in most of the cases, the `CM` version outperforms the `RM` version. This is because the loop nests have originally been extracted from Fortran programs. In `transpose.1` and `vpenta.6`, the `UN` and `DT` versions behave the same and outperform the other versions in particular for the cache sizes less than 128K. The reason for this can be seen from Table 1; these are the only loop nests without any temporal reuse. Also due to conflicting accesses to different arrays, the `LP` version does not help with these nests either. The only remaining possibility is using the data layout transformations derived by the `UN` and `DT` versions. This result shows that for the loop nests without temporal reuse, our approach achieves the same result as pure data transformations.

For the loop nests with very high temporal reuse (`htribk.2`, `qzhes.7` , and `mxm.2`) the relative success of loop and data transformations depends largely on the number of dimensions of the arrays with temporal reuse as well as the degree of conflict in optimal layout requirements between different references. In our cases, the data transformations performed very well for `htribk.2` and `qzhes.7` as the arrays with temporal reuse are two-dimensional and there are no conflicts in optimal layout requirements. Since `mxm.2` has already been optimized through loop unrolling for the fixed column-major layouts, the `DT` and the `RM` versions kind of reverse the optimizations. It should be noted that although pure data transformations may not be successful for the loop nests with high temporal reuse, our approach (`UN`) will be successful in general as it first optimizes for temporal locality via loop transformations. For `hydro2d/T1.1`, `emit.4`, `bakvec.2`, `gfunp.4`, and `rhf_hessian.1`, the programmers did a good job as far as the locality is concerned. For these codes, `CM`, `LP`, `DT`, and `UN` behave similarly. The `RM` version however performs very poorly as the column-major layouts fit very well with the existing loop structures in those codes. It should be emphasized however, the `UN` version still offers some improvement in these codes over the `CM`, `LP`, and `DT` versions.

As for the remaining loops, `btrix.4` has a structure that allows any locality optimization method (even a simple conversion to row-major layouts) to improve its performance. The `UN` version, however, improves it the most by first taking advantage of temporal locality and then using layout transformations for two four-dimensional arrays. A similar situation also occurs with `fnorm.1`, `r1mpyq.1`, and `rhf_hessian.2`. For `hydro2d/fct.7` code, the `CM`, `LP`, `DT` and `UN` versions result in exactly the same transformed code. For the `adi.2` code, which contains the maximum number of references, again the `UN` version outperforms the others and results in the same code as `LP`.

Overall the `UN` version seems to be quite successful and consistently achieves in general the best results obtained by our experiments. In the loop nests where there is no temporal reuse, our approach can optimize spatial locality using data transformations. For the loop nests with high temporal reuse, our approach first optimizes potential temporal reuse using loop transformations, then uses data transformations for the remaining references. Even in the programs where the original locality is good, we can still offer a marginal improvement. Our approach works for those cases in which loop transformations are preferable to data transformations or vice-versa and takes advantage of that through an integrated approach.

5.3 Execution time results

The experimental platform that we used to evaluate our locality optimization method is a single node of the SGI Origin 2000 dis-

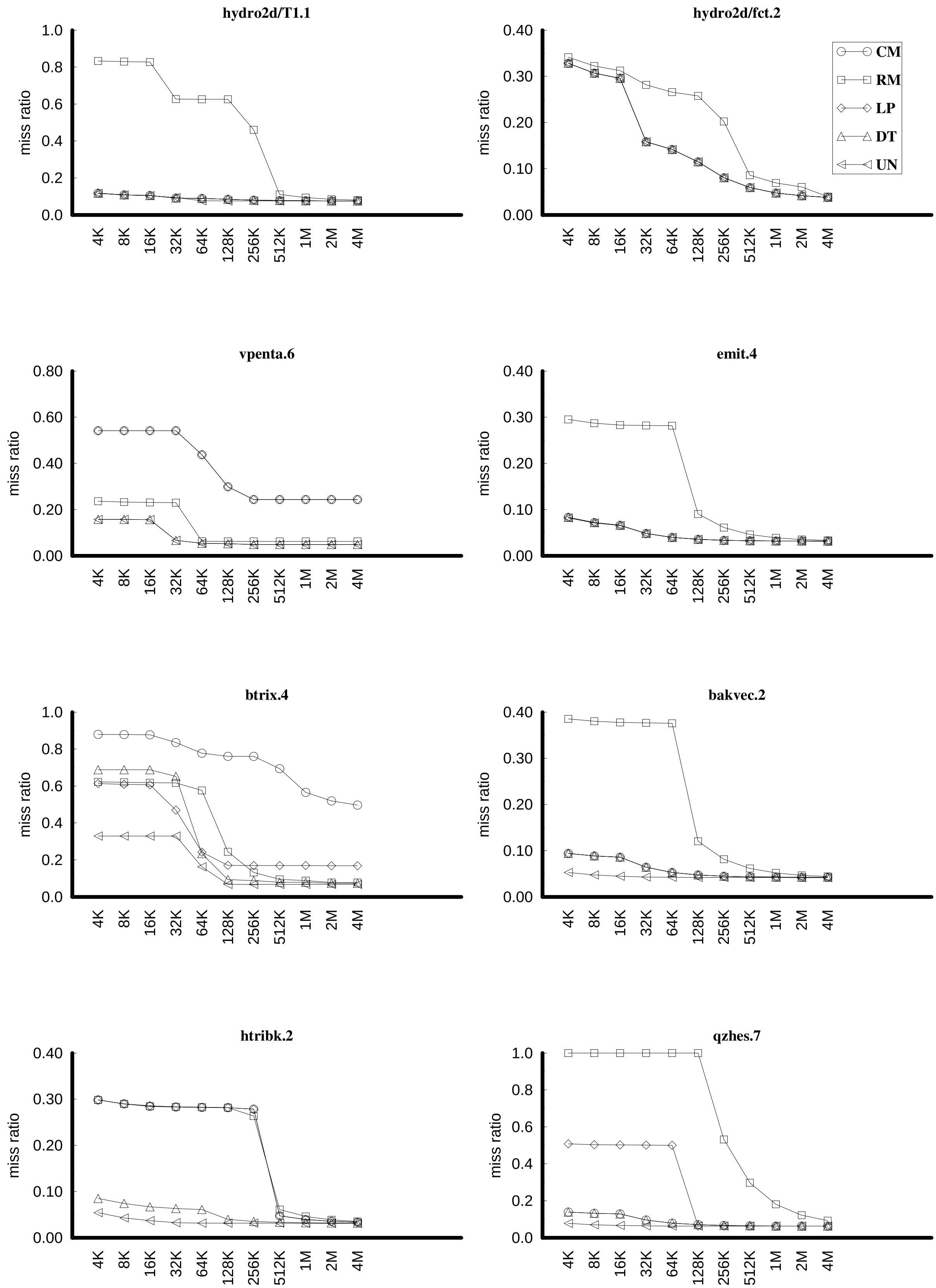

Figure 6: Miss rates with varying cache sizes for 8 benchmarks.

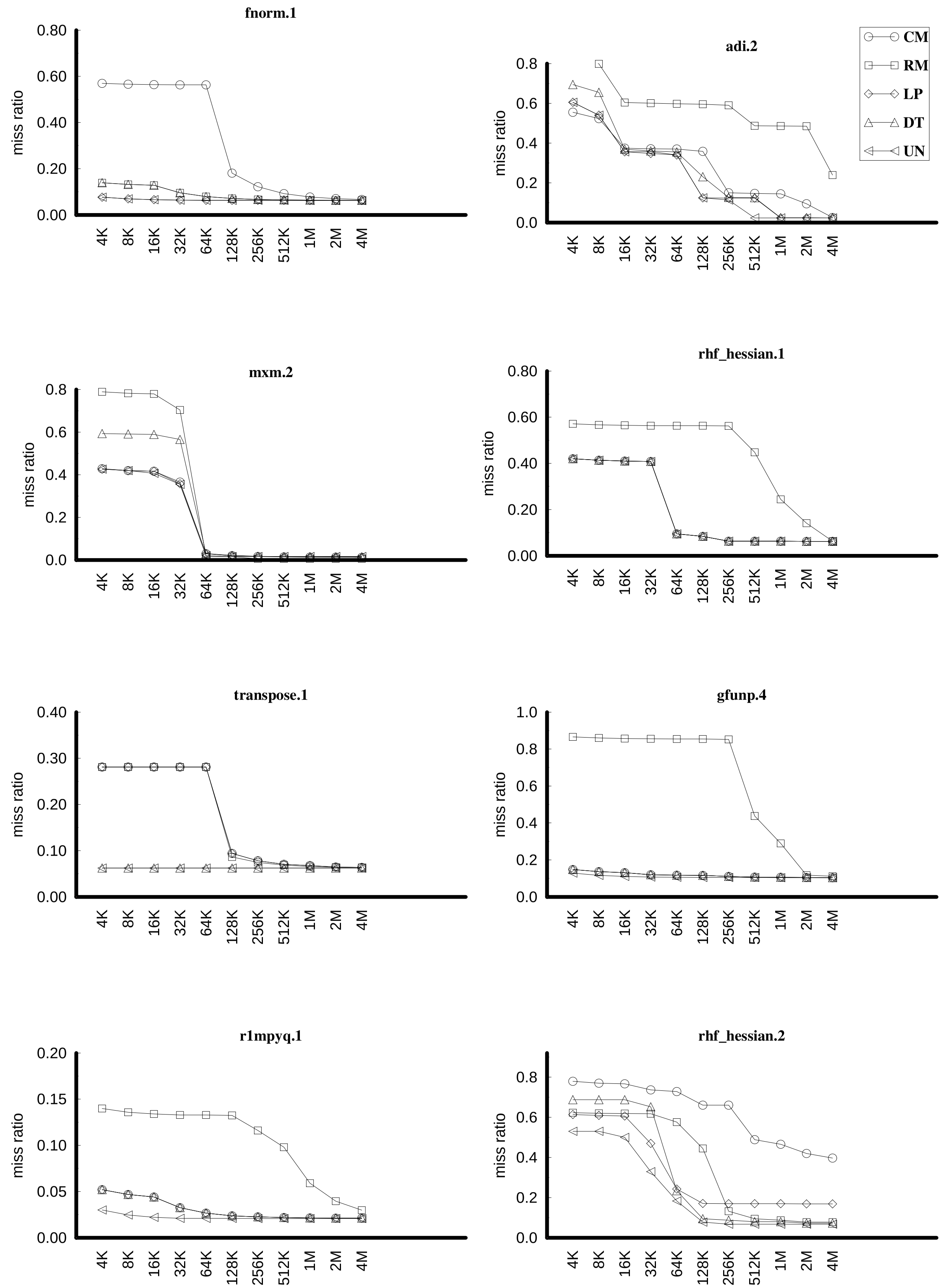

Figure 7: Miss rates with varying cache sizes for other 8 benchmarks.

tributed shared memory multiprocessor. The node uses 195MHz R10000 MIPS processor, with 32KB L1 data cache and 4MB L2 unified cache. R10000 is an advanced superscalar processor which can fetch and decode four instructions per cycle and can run them on five pipelined functional units. Both caches are two-way associative and non-blocking. Up to 4 outstanding misses from the combined two levels of cache are supported. R10000 dynamically schedules instructions whose operands are available in order to hide the latency of cache misses. For L1 cache hits, the latency is 2 cycles; and for L1 misses that hit in L2, the latency is 8 to 10 cycles. The C versions of the programs are compiled using the native `C` compiler using `-O2` option.

In the experiments we used two input sizes for each loop nest: `small` and `large`. Except for a few codes, each array dimension is set to $2,048$ for `small` and to $4,096$ for `large`. Figure 8 shows the execution times for the `small` input sizes. The times are normalized with respect to the worst performing version. These results in general are consistent with the miss rates and show that except for one case, the `UN` version achieves the best performance, the same result obtained by the simulation. However, due to small differences between miss rates, the relative performances of the other versions differ in some cases from the miss rates. Similar results are also observed with the large inputs in Figure 9. Notice that when the input size is increased, in general the effectiveness of the `UN` version also increases.

5.4 Discussion

Table 3 shows the summary of simulation results for our loop nests. If a version x achieves the best measured miss rate for a loop nest y, we put a `+` symbol in location (x, y). In obtaining this table, we only considered the cache sizes up to 64K. It is easy to note that except for one case, the `UN` version always resulted in the best miss rates. In contrast, the `DT` and the `LP` versions achieved the best rates in 5 and 4 times respectively. In 9 of the 16 loop nests neither `DT` nor `LP` obtained the best rates. These results show that neither `DT` nor `LP` dominates the other; and our technique consistently outperforms both of them and can lead to significant improvements in miss rates.

It should be noted that as far as the `UN` version is concerned, the execution times also follow this trend for our benchmark nests. However, the performance of the miss rates and that of the execution time can differ for other versions due to the large amount of overlapped activity in the processor.

Although our main objective is to improve cache locality using an integration of loop and data transformations, as mentioned earlier, since our technique associates different types of locality with each reference, it can provide useful hints for the new cache architectures.

6 Related work

Much of the related compiler work to cache locality optimization is based on iteration space transformations. Wolf and Lam [21] define reuse vectors and reuse spaces, and show how these concepts can be exploited by an iteration space optimization technique. Their approach first finds the loop levels which carry reuse and then uses unimodular loop transformations to bring the loops carrying reuse in the innermost positions. In this step they use a kind of exhaustive search. Afterwards, they apply tiling to the loops that carry some form of reuse. Li [17] also uses reuse vectors to detect the dimensions of loop nest which carry reuse. Instead of resorting to exhaustive search, however, he determines the appropriate loop transformation matrix in one shot. In contrast, Carr et al. [4] use a simple locality criterion to reorder the computation to enhance data locality. The locality improving techniques also include blocking [4, 6, 11, 14, 15, 21].

All these techniques however focus on the iteration space and attempt to improve data locality indirectly by modifying the loop access patterns. They are not easily applicable to programs with imperfectly-nested loops (except [14]) and are constrained by data dependences. We, instead, demonstrated in this paper that data transformations can also play a significant role in enhancing cache locality.

More recently new techniques based on memory layout transformations have been proposed. These techniques focus directly on array layouts and try to modify the layouts such that unit-stride accesses will be obtained in the innermost loop. O'Boyle and Knijnenburg [19] explain how to generate code given a data transformation matrix. They show different usages of data transformations in addition to optimizing spatial locality. Leung and Zahorjan [16] focus more on minimizing memory consumption after a layout transformation. These two techniques however are oriented toward exploiting spatial reuse rather than temporal reuse as they are based on pure data transformations. More importantly, since the impact of a layout change may penetrate into multiple nest, the scope to be considered becomes much larger. In comparison, our approach uses loop transformations to minimize the adverse effect of a layout transformation in another nest. Since we give priority in optimizing temporal locality, our approach can potentially be more successful than a pure data layout based approach.

Cierniak and Li [5] were among the first to offer a scheme that combines loop and data transformations. There are a number of differences between their work and ours. First, we use general unimodular loop and data transformations whereas they restrict the search space for possible loop and data transformations. For example, their framework cannot convert a memory layout from column-major to diagonal. Second, their optimization methodology is centered around a concept called the *stride vector* whose entries should be guessed by compiler before the optimization process. In contrast, we do not have such a requirement. Third, they focus largely on a single loop nest. Their extension to multiple loop nests is not very clear whereas we can handle multiple loop nest case using layout constraints. Lastly, we give priority to optimizing temporal locality whereas they do not distinguish between different types of locality. Kandemir et al. [13] have also considered loop and data layout optimizations. Like Cierniak and Li [5], their solution restricts the search space of data transformations to dimension permutations; in addition, theirs is an exhaustive search technique. Anderson et al. [1] propose a transformation technique that make data elements accessed by the same processor contiguous in the shared address space. They use only permutations of array dimensions and strip-mining for possible data transformations. Our work is more general as we consider a much larger search space for possible layout transformations.

7 Conclusions

This paper describes an integrated compiler approach to enhance cache locality. Our approach combines loop and data transformations, but specializes the loop transformations for optimizing temporal locality. Once the potential temporal reuse is exploited, our approach uses data transformations to optimize available spatial reuse in the nest. We also show how our technique can be extended to work with cases in which some subset of the arrays referenced in the nest have fixed memory layouts. We discuss the great importance of this extension for optimizing cache locality globally (i.e., program-wide). The information required by the compiler to apply our technique is easily obtained during dependence analysis, which is performed by almost every optimizing compiler. Once this information is obtained, our approach uses simple linear algebraic tech-

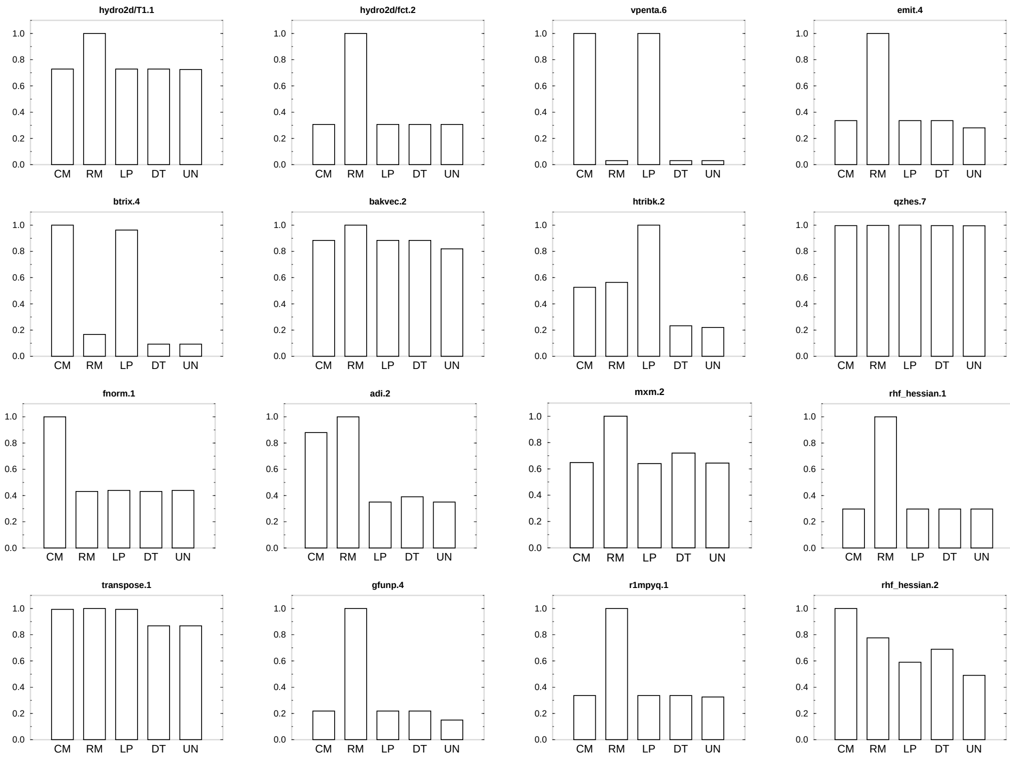

Figure 8: Normalized execution times with `small` input sizes.

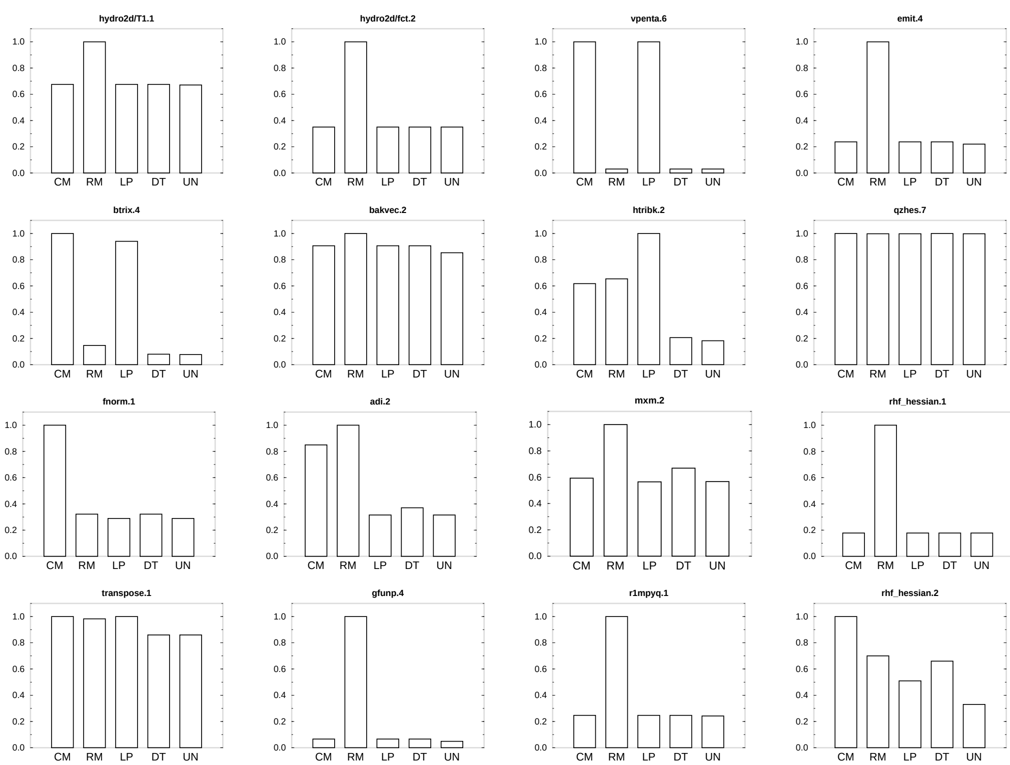

Figure 9: Normalized execution times with `large` input sizes.

Table 3: Summary of the results. If a version x achieves the best measured miss rate for a loop nest y, we put a + symbol in location (x, y). In obtaining these results, only the cache sizes up to 64K are considered.

VERSION	LOOP NEST #															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
CM		+										+				
RM											+					
LP		+							+	+		+				
DT		+	+	+								+	+			
UN	+	+	+	+	+	+	+	+	+	+		+	+	+	+	+

niques to manipulate loop nests, array references and array declarations. Our simulation results demonstrate that our technique does make a difference, and improves over techniques that are based on pure loop or pure data layout transformations. Our preliminary execution time results also show the effectiveness of our approach.

This work can be seen as a first step toward combining data and loop transformations in an integrated framework. It integrates linear loop and linear data transformations. An important future direction is to investigate the interaction of data layout transformations with tiling, another loop-based transformation technique. It is known that applying locality-enhancing loop transformations before tiling is preferable as it makes performance of the tiling less sensitive to the tile size [5, 17]. The impact of integrating loop and data transformations on tiling merits further investigation. Another important question is whether or not storing the elements accessed by a tile in consecutive memory locations will further help. This is also a data transformation although not linear. We are studying those cases where such a data transformation might be useful. In addition, we plan to enhance our technique by taking into account the effects of instruction scheduling and register allocation.

Acknowledgments

The work of M. Kandemir and A. Choudhary was supported in part by NSF Young Investigator Award CCR-9357840, NSF grant CCR-9509143 and Air Force Materials Command under contract F30602-97-C-0026. J. Ramanujam was supported in part by NSF Young Investigator Award CCR-9457768. P. Banerjee was supported in part by DARPA under contract F30602-98-2-0144 and by the NSF grant CCR-9526325.

References

[1] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

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

[3] A. Bik, and H. Wijshoff. On a completion method for unimodular matrices. Technical Report 94-14, Dept. of Computer Science, Leiden University, 1994.

[4] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proc. the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

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

[6] S. Coleman, and K. McKinley. Tile size selection using cache organization and data layout. In *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[7] C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. A strategy for array management in local memory. In *Proc. the 3rd Annual Workshop on Languages and Compilers*, August 1990.

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

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

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

[11] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.

[12] 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 International Conference on Supercomputing*, 1998.

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

[14] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. Programming Language Design and Implementation (PLDI)*, June 1997.

[15] M. Lam, E. Rothberg, and M. Wolf. The cache performance of blocked algorithms. In *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, April 1991.

[16] S-T. Leung, and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR 95-09-01, CSE Dept., University of Washinton, 1995.

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

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

[19] M. O'Boyle, and P. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers (CPC 96)*, pages 287–297, Aachen, Germany, 1996.

[20] A. Schrijver. *Theory of linear and integer programming*, John Wiley, 1986.

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

[22] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proc. MICRO-29,* pages 274–286, Paris, France, December 1996.

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