A Matrix-Based Approach to the Global Locality Optimization Problem

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

Abstract

Global locality analysis is a technique for improving the cache performance of a sequence of loop nests through a combination of loop and data layout optimizations. Pure loop transformations are restricted by data dependences and may not be very successful in optimizing imperfectly nested loops; the impact of a data transformation on an array might be program-wide. Therefore, in this paper we argue for a combined approach which employs both loop and data transformations. The method enjoys the advantages of the most of the previous techniques for enhancing locality and is efficient. In our approach, the loop nests are processed one by one and the data layout constraints obtained from one nest are propagated for optimization of the remaining loop nests. We show that this process can be put in a simple matrix framework which can be manipulated by an optimizing compiler. The search space that we consider for possible loop transformations comprises general non-singular linear transformation matrices and the data layouts that we consider are those which can be expressed using hyperplanes. Experiments using several programs on an SGI Origin 2000 distributed-shared-memory machine demonstrate the efficacy of our approach.

1 Introduction

A key component of the success of high performance computing is the support for software development without undue concern on the users' part regarding the details of the underlying hardware. It is difficult for users to exploit the performance potential of current machines since a careful orchestration of the program appears necessary. Due to technological advances, processor speeds have improved at a much higher rate than memory speeds. In fact, during the last decade the processor speeds have been improving at a rate of at least 50% each year while memory access time has improved only at the rate of 7% per year, resulting in a significant widening of the performance gap between processors and memory subsystems [5]. Modern multiprocessors include several levels of memory hierarchy in which the lower levels are slow and inexpensive (e.g., disks, memory) while the higher levels (e.g., caches, registers) are fast but expensive.

Fortunately, compiler technology has played a key role by enabling the restructuring of programs in order to exploit the architectural features of modern machines, thus relieving some of the burden on the users. The role of the compiler has become increasingly significant due to the rapid evolution of parallel machine architectures. A recent study shows that a group of highly parallelized scientific benchmark programs spend as much as a quarter to a half of their execution times waiting for data from memory [17]. In order to eliminate the memory bottleneck, cache locality should be exploited as much as possible. One way of achieving this is to transform loop nests to improve locality. There has been a great deal of research in the area of loop transformations. Several loop transformations such as interchange, skewing, etc. have been incorporated into a single framework using a matrix representation of transformations [20]. Among these techniques used are unimodular and non-unimodular iteration space transformations and tiling [19, 18]. There are two common characteristics of these techniques: (1) they focus on improving data locality indirectly as a result of modifying the iteration space traversal order; and (2) a transformation is applied to one loop nest at a time. An important drawback of loop transformations is that they are constrained by data dependences.

Proper layout of data in memory may also have a significant impact on the performance of scientific computations on multiprocessor machines. In this paper, we demonstrate techniques that decide good data layouts (e.g., row-major or column-major storage of matrices) and appropriate loop transformations that together improve the performance of dense matrix codes significantly. We refer to the optimization of data layouts as *data transformations.* Unlike loop transformations, data layout optimizations are not constrained by dependences. We assume that every array accessed in a program has a single fixed memory layout for the whole program. As a result, data layout decisions affect the performance characteristics of the whole program unlike loop transformations; in addition, data layouts impact the choice of loop transformations applied to each loop nest. We model data transformations using matrices in the same way loop transformations have been modeled. Such an approach allows us to exploit the benefits of a unified linear algebraic framework.

Only a few papers have considered data transformations to optimize locality. O'Boyle and Knijnenburg [15] focus on restructuring the code given a data transformation matrix; they show that their method can be used for optimizing spatial locality. In comparison, we concentrate on the problem of determining suitable layouts *together* with loop transformations, so that the whole program can be optimized. Anderson, Amarasinghe and Lam [1] propose a data transformation technique for distributed shared memory machines. While they restrict themselves to strip-mining and permutation alone, we consider the full range of layout transformations that are expressible using hyperplanes. Finally, Cierniak and Li [3] consider only dimension re-indexing (permutation-based layout transformations) and use a pseudo-exhaustive search to detect array layouts. In contrast to [3] which restricts possible loop as well as data transformations, the approach explained in this paper considers a much larger search space for memory layouts and loop transformations.

This paper is organized as follows. Section 2 presents an outline of the global locality optimization problem. Sections 3 and 4 present the details of our approach. Section 5 presents preliminary experimental results on SGI Origin 2000 distributed-shared-memory machine. Section 6 reviews related work on locality op-

*Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244. e-mail: `mtk@ece.nwu.edu`

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

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

timization and Section 7 summarizes the paper with an outline of ongoing and future work.

2 Overview of our approach to global locality optimization

As noted earlier, there is a vast amount of work in optimizing locality for a single loop nest using iteration space (loop) transformations [18, 12, 13]. Recently some authors [1, 15, 11] have proposed techniques to optimize spatial locality in a loop nest using data space transformations. Such a transformation typically modifies the memory layout of a multi-dimensional array. The main problem with this approach is that modifying the memory layout of an array has a *global* effect meaning that it affects the locality of all references to that array in all the loop nests. We show that such a global impact can in fact be exploited using a proper mix of data space and iteration space transformations.

Our approach to the global locality optimization problem can be defined informally as follows. First, we transform the program into a canonical form using loop fusion, loop distribution, and code sinking [20]; in this canonical form, the program contains two types of references, the ones that occur inside loop nests, and those that occur between loop nests. In our approach, only the references within loop nests can affect memory layout decisions; in determining layouts, we simply ignore the references between loop nests. Then we construct an interference graph similar to that used by Anderson and Lam [2]. This is a bipartite graph which contains two sets of vertices; one set corresponding to the loop nests and the other corresponding to the arrays. There is an undirected edge between an array vertex and a loop nest vertex if and only if that loop nest accesses that array. Our technique works on a single connected component of this graph at a time, since there are no common arrays between different connected components.

For a single connected component, we first order the loop nests according to a cost criterion, from the most costly nest to the least costly. Practically, the *cost* of a loop nest can be defined as its uniprocessor execution time. We currently use profile information to compute the cost of each nest in a given connected component. Profiling the sequential code also allows us to obtain array sizes, loop bounds and probabilities of conditional statements. Notice that this ordering of loop nests is not a modification or transformation to the structure of the program; but rather a step to determine in which order the loop nests will be considered by our algorithm. Then the algorithm starts with the most costly nest and optimizes it for locality. After this process, suitable memory layouts for (possibly some of) the arrays referenced in this loop nest are determined. Afterwards, we move to the next most costly nest. In optimizing this nest, we take into account the memory layouts determined during the optimization of the most costly nest. After each nest is optimized, (possibly) new memory layouts are determined and all the memory layouts obtained so far are propagated as layout constraints for optimizing the next nest in the connected component. During the processing of a single connected component, when a loop nest is to be optimized, the compiler can encounter three different possibilities: (1) No constraint exists; that is, none of the arrays have a determined memory layout so far, and we have complete freedom on the choice of loop transformations. In this case, the compiler's task is to determine the memory layouts of the arrays referenced in the loop nest as well as to find an accompanying iteration space transformation; (2) The memory layouts of some of the arrays referenced in the loop nest have already been fixed during the processing of a previous loop nest. Here, the task is to determine the memory layouts of the remaining arrays and to transform the iteration space accordingly; and (3) We have restrictions in the iteration space other than those due to data dependences. These restrictions are in general related to parallelism; typically, these restrictions can arise due to the need to reduce (or perhaps eliminate) false sharing [16] or the need to exploit the largest granularity of parallelism [18]. Of course, Case 1 can be seen as a special form of Cases 2 or 3 and a combination of Cases 2 and 3 as well as several other special cases may also occur. In the next section we give details of our approach.

3 Data and loop transformations

3.1 Canonical form

We assume that the only control flow that we have is loop structures. In our canonical form, a program consists of only a sequence of (preferably perfectly-nested) loop nests. We bring a program into this form using a technique similar to that proposed by McKinley et al. [13], which uses loop fusion and loop distribution.

Our algorithm first normalizes each nest such that the step size of each loop becomes one. It then focuses on imperfectly nested loops and transforms them into perfectly nest loops using a combination of loop fusion, loop distribution and code sinking. After a sequence of independent loop nests is obtained, a final pass applies loop fusion once more combining adjacent loop nests if doing so improves temporal locality without causing undue register pressure. The overall approach is similar to that proposed in [13]; the details are beyond the scope of this work.

Consider the program fragment shown on the left part of Figure 1. This fragment consists of two imperfectly nested loop nests. Within the loop nests are the names of the arrays accessed. Our approach transforms these loop nests into a series of perfectly nested loops. In this example, we assume that this can be accomplished using loop fusion for the first imperfectly nested loop nest and loop distribution for the second.

3.2 Interference graph

Next the compiler builds an interference graph similar to that used by Anderson and Lam [2] in automatic data decomposition. This is a bipartite graph (V_n, V_a, E) where V_n is the set of loop nests, V_a is the set of arrays, and E is the set of edges between loop vertices and array vertices. There is an edge $e \in E$ is between $v_a \in V_a$ and $v_n \in V_n$ if and only if v_n references v_a. Then we run a connected-component algorithm on this graph. For the example given in Figure 1, we have two connected components. Each connected component is fed into our global locality optimization algorithm explained in the rest of this paper.

3.3 Hyperplane-based layout representation

Our approach to memory layout representation is based on hyperplane theory from linear algebra and is briefly explained below. The details are beyond the scope of this paper and can be found elsewhere [7]. In this framework, hyperplanes are used to represent memory layouts of multi-dimensional arrays. For an m-dimensional array, a hyperplane defines a set of array elements $(\jmath_1, \jmath_2, \cdots, \jmath_m)$ which satisfy the following equation

$$g_1\jmath_1 + g_2\jmath_2 + \cdots + g_m\jmath_m = c \quad (1)$$

where c is a constant. In this equation, $g_1, g_2, \cdots, g_m$ are rational numbers called hyperplane coefficients and c is a rational number called hyperplane constant [6]. We refer to $\bar{g} = (g_1, g_2, \cdots, g_m)^T$ as a hyperplane vector associated with equation (1). When there is no ambiguity, all transposition symbols from vectors will be omitted. A *hyperplane family* is a set of hyperplanes with the same coefficients but with a different constant (c value).

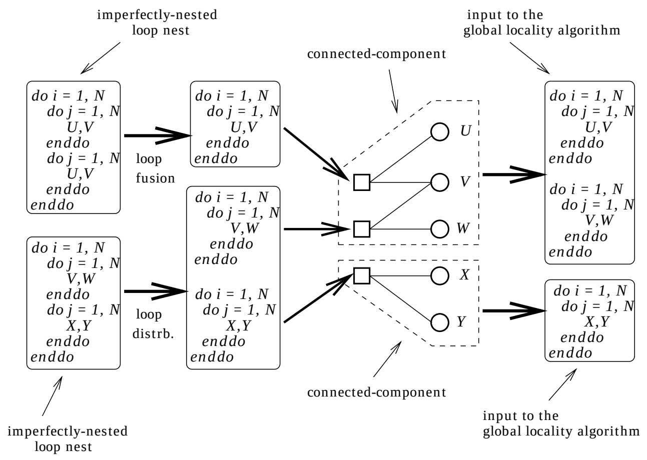

Figure 1: Example application of locality optimization algorithm.

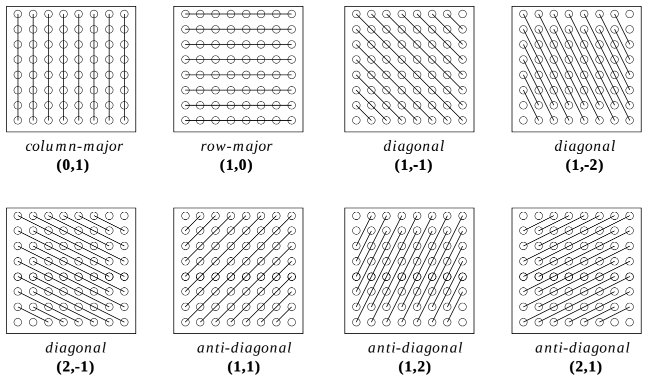

Figure 2: Example memory layouts for two-dimensional arrays and their hyperplane vectors.

An important observation is that a hyperplane family can be used to partially define the memory layout of a multi-dimensional array. Let us concentrate now on a two dimensional $N \times N$ array stored in column-major form in memory as is the case in Fortran. We can think of each column of this array as a hyperplane (a line); and all columns collectively define a hyperplane family. Here, the hyperplane vector is $(0, 1)$ and hyperplane equation is $\jmath_2 = c$ where $1 \le c \le N$. For example, $\jmath_2 = 5$ represents the fifth column of the array. Two array elements $\bar{J}$ and $\bar{J}'$ belong to the same hyperplane $\bar{g}$ if

$$\bar{g}\bar{J} \quad = \quad \bar{g}\bar{J}'. \tag{2}$$

Returning to our two-dimensional column-major array, since the array elements $(1, 5)$ and $(4, 5)$ satisfy equation (2) for $\bar{g} = (0, 1)$ they belong to the same hyperplane which can be identified with $c = 5$.[1] On the other hand, for instance, $(1, 5)$ and $(1, 6)$ do not satisfy equation (2), therefore, they belong to different hyperplanes. It is important to stress that the memory layouts defined by hyperplanes are not limited to the conventional layouts such as column-major and row-major. For example, a hyperplane family defined by $(1, -1)$ also represents a memory layout where, say, the array elements $(2, 3)$ and $(4, 5)$ map on the same hyperplane. It is easy to see that such a memory layout corresponds to diagonal-layout (or skewed-layout) where the elements in each diagonal are stored consecutively in memory. Similarly, the hyperplane vectors given by $(1, 0)$ and $(1, 1)$ correspond to row-major and anti-diagonal memory layouts respectively. Figure 2 shows a few possible memory layouts for an 8×8 array and the associated hyperplane vectors. Each circle in this figure represents an array element. With such a representation, we say that two array elements have *spatial locality* (or physical proximity) if they belong to the same hyperplane. For example, $(1, 5)$ and $(4, 5)$ have spatial locality in column-major layout whereas $(2, 3)$ and $(4, 5)$ have spatial locality in diagonal-layout expressed using $(1, -1)$. Notice that for example in column-major memory layout, our spatial locality definition does not encompass two elements that are mapped onto different columns but in consecutive memory locations.

For two-dimensional arrays, a single hyperplane family is sufficient to define the 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: $\bar{g} = (0, 0, 1)$ and $\bar{h} = (0, 1, 0)$. We can write these two hyperplanes collectively as a layout constraint matrix or simply a layout matrix

$$L_U = \begin{pmatrix} \bar{g} \\ \bar{h} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

In that case, two data elements $\bar{J}$ and $\bar{J}'$ have spatial locality (i.e., map onto the same hyperplane) if both of the following are true:

$$\bar{g}\bar{J} \quad = \quad \bar{g}\bar{J}' \tag{3}$$
$$\bar{h}\bar{J} \quad = \quad \bar{h}\bar{J}' \tag{4}$$

The elements that exhibit spatial locality should be stored in consecutive memory locations. This idea can easily be generalized to higher dimensions as well [7].

3.4 Single nest-level optimizations

Nest-level optimizations (or local optimizations) transform a loop nest to increase cache locality. Essentially, the objective is to obtain either temporal locality or stride-one access of the arrays which is important for architectures with some form of cache hierarchy. To understand the effect of a loop transformation let us represent a loop nest of depth n which consists of loops $i_1, i_2, \cdots, i_n$ as a polyhedron defined by the loop limits. We use an n-dimensional vector $\bar{I} = (\imath_1, \imath_2, \cdots, \imath_n)$ called the iteration vector to denote the execution of the body of this loop nest with $i_1 = \imath_1, i_2 = \imath_2, \cdots, i_n = \imath_n$.

We assume that the array subscript expressions and loop bounds are affine functions of enclosing loop indices and symbolic variables. We can model each array reference using an access matrix $\mathcal{L}$ and a constant vector $\bar{o}$ [18, 12]. As an example, a reference $U(i + 1, j)$ to a two-dimensional array U in a loop nest of depth two with i as the outer loop index is represented by $\mathcal{L}\bar{I} + \bar{o}$, where

$$\mathcal{L} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ and } \bar{o} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

In general, if the loop nest is of depth n and the array in question is m-dimensional, the access matrix is of size $m \times n$ and the constant vector is m-dimensional.

The class of iteration space transformations we are interested in can be represented using linear non-singular transformation matrices. For a loop nest of depth n, the iteration space transformation

[1] Notice that the multiplies in equation (2) are dot products.

matrix T is of size $n \times n$. Such a transformation maps each iteration vector $\bar{I}$ of the original loop nest to an iteration $\bar{I}' = T\bar{I}$ of the transformed loop nest. Therefore, after the transformation, the new subscript function is $\mathcal{L}T^{-1}\bar{I}' + \bar{o}$. The problem investigated in works such as [18] and [12] is to select a suitable T such that the locality of the reference is improved and all the data dependences in the original nest are preserved.

Consider the three-point stencil computation nest shown in Figure 3(a). Assuming column-major memory layouts for U and V, the accesses to both the arrays are poor from the locality point of view. The problem is that successive iterations of the inner loop j touches different columns. The chances are very low that a line brought into cache in one of these iterations will stay in the cache when any of its elements is reused. An iteration space transformation technique such as the one proposed by Li [12] optimizes this nest by interchanging [20] the loop order which is legal in this case. This loop transformation can be represented by a *unimodular* transformation matrix

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

We note that the same nest can also be improved using data transformations instead. Our approach uses the hyperplane-based layout representation explained earlier.

Let $\bar{I} = (i, j)$ and $\bar{I}_{next} = (i, j+1)$; that is, $\bar{I}$ and $\bar{I}_{next}$ are two consecutive iterations. We focus on array U, a similar analysis also applies to array V. Two data elements accessed by $\bar{I}$ and $\bar{I}_{next}$ are $\mathcal{L}\bar{I}+\bar{o}$ and $\mathcal{L}\bar{I}_{next}+\bar{o}$ respectively. Using equation (2), in order to have a spatial locality $\bar{g}(\mathcal{L}\bar{I}+\bar{o}) = \bar{g}(\mathcal{L}\bar{I}_{next}+\bar{o})$ should be satisfied, where $\bar{g}$ represents an optimal layout. Taking into account $\bar{I}$ and $\bar{I}_{next}$, solving this last equality gives us $\bar{g}\bar{\ell} = 0$ where $\bar{\ell}$ is the last column of $\mathcal{L}$. That is, if we choose a hyperplane vector $\bar{g}$ such that $\bar{g} \in Ker\{\bar{\ell}\}$, we will have spatial locality. Since, in our example,

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

we have $\bar{\ell} = (0,1)^T$. Choosing $\bar{g}$ from null space of $\bar{\ell}$, gives us $\bar{g} = (1,0)$, which, as mentioned earlier (see Figure 2), corresponds to row-major memory layout. The details of how to select a suitable vector from Ker set (null set) are not important for the purposes of this paper. In summary, in order to have a good spatial locality in the innermost loop, we have to change the memory layout of U (and that of V) from column-major to row-major.

4 Combined data and loop transformations for improving locality

We have shown in the previous section that in order to optimize spatial locality of a loop nest both loop and data transformations may be used. In the following, we show how to combine these two optimization techniques. Let us focus on a two-dimensional array U referenced in a loop nest of depth two using an access matrix $\mathcal{L}$. The results to be presented easily extend to higher dimensional arrays and loop nests as well. We define $\bar{I}' = (i, j)$ and $\bar{I}'_{next} = (i, j+1)$ as two consecutive iteration vectors *after* the transformation. Also assume that we will use a 2×2 transformation matrix T and $Q = T^{-1}$. After the transformation, in order to have spatial locality (see equation (2)),

$$\bar{g}(\mathcal{L}Q\bar{I}' + \bar{o}) = \bar{g}(\mathcal{L}Q\bar{I}'_{next} + \bar{o})$$

should be satisfied. Solving this equation, we obtain

$$\bar{g}\mathcal{L}\bar{q} \quad = \quad 0, \qquad (5)$$

where $\bar{q}$ is the last column of Q. Therefore the problem is to find a $\bar{g}$ and a $\bar{q}$ for a given $\mathcal{L}$ such that the equation given by (5) will be satisfied.

Notice that this equation is non-linear and is with regard to a single loop nest and a single reference. In order to optimize locality globally (program-wide) we should set up and solve simultaneously the equations similar to (5) for each reference in each loop nest. Of course, given a large number of loop nests and references, this system of equations may not have a solution, in which case we should ignore some equations. Exactly which equations will be ignored depends largely on the profile information.

We illustrate the process using an example first. Consider the program fragment given in Figure 3(b). The access matrices for this program are as follows:

- For the first nest: $\mathcal{L}_U = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \mathcal{L}_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.
- For the second nest: $\mathcal{L}_{U_1} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \mathcal{L}_{U_2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

We would like to find suitable loop transformation matrices for both the loop nests, and to determine accompanying memory layouts for arrays U and V. Let T and S denote the transformation matrices for the first and second nest respectively, and $Q = T^{-1}$ and $R = S^{-1}$. Also let $\bar{q} = (q_{12}, q_{22})^T$ and $\bar{q} = (r_{12}, r_{22})^T$ be the last columns of Q and R respectively. Let $\bar{g} = (\alpha_g, \beta_g)$ and $\bar{h} = (\alpha_h, \beta_h)$ represent the optimal layouts of U and V respectively. Using (5), we obtain the following for the first loop nest:

- For array U: $(\alpha_g, \beta_g)\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} q_{12} \\ q_{22} \end{pmatrix} = 0$.
- For array V: $(\alpha_h, \beta_h)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} q_{12} \\ q_{22} \end{pmatrix} = 0$.

For the references to array U in the second nest, $(\alpha_g, \beta_g)\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} r_{12} \\ r_{22} \end{pmatrix} = 0$ and $(\alpha_g, \beta_g)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} r_{12} \\ r_{22} \end{pmatrix} = 0$. We can write these equations collectively as

$$\begin{pmatrix} q_{12}+q_{22} & q_{22} & 0 & 0 \\ 0 & 0 & q_{12} & q_{22} \\ r_{22} & r_{12} & 0 & 0 \\ r_{12} & r_{22} & 0 & 0 \end{pmatrix}\begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \end{pmatrix} = \bar{0}. \qquad (6)$$

It should be noted that a solution to such a system for q_{12}, q_{22}, r_{12}, r_{22}, α_g, β_g, α_h, and β_h will give us suitable loop transformation matrices (actually only last columns of the inverses of them) as well as optimized memory layouts. However, we have some additional constraints as well; specifically, for each unknown vector such as $\bar{q}$, $\bar{r}$, $\bar{g}$ and $\bar{h}$ at most one of the entries may be zero. With these additional constraints solving the non-linear system given by equation (6) is very difficult.
What we need is a heuristic that works fast in practice and generates acceptable near-optimal solutions. We note that (6) can easily be divided into two sub-matrix equations each for a single nest:

$$\text{Nest 1: } \begin{pmatrix} q_{12}+q_{22} & q_{22} & 0 & 0 \\ 0 & 0 & q_{12} & q_{22} \end{pmatrix}\begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \end{pmatrix} = \bar{0} \qquad (7)$$

```
do i = 3, N
  do j = 3, N
    U(i,j)=(V(i-2,j)+V(i,j)+V(i,j-2))/3.0
    end do
end do
```
(a)

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

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

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

do u = -N+1, N-1
  do v = max(1-u,1), min(N-u,N)
    ···=U(v,u+v)+U(u+v,v)
  end do
end do
```
(c)

Figure 3: (a) An example loop nest. (b) Original fragment. (c) Optimized version of (b).

$$\text{Nest 2:} \quad \begin{pmatrix} r_{22} & r_{12} & 0 & 0 \\ r_{12} & r_{22} & 0 & 0 \end{pmatrix} \begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \end{pmatrix} = 0. \qquad (8)$$

There is a coupling between (7) and (8) due to layout vectors. Let us first focus on (7), and assume that this loop nest is more costly than the second one and no iteration space transformation will be applied; that is, Q is the identity matrix meaning that $q_{12} = 0$ and $q_{22} = 1$. Later in the paper we discuss this decision in detail. We can now think of (7) in a block form as shown below:

$$\left(\begin{array}{cc|cc} q_{12} + q_{22} & q_{22} & 0 & 0 \\ \hline 0 & 0 & q_{12} & q_{22} \end{array}\right) \begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \end{pmatrix} = \bar{0} \qquad (9)$$

This last equation can be written symbolically as follows:

$$\begin{pmatrix} S_I & 0 \\ 0 & S_{II} \end{pmatrix} \begin{pmatrix} \bar{g} \\ \bar{h} \end{pmatrix} = \bar{0}$$

where S_I and S_{II} correspond to non-zero sub-matrices in (9). Now we need to solve two equations: $S_I\bar{g} = 0$ and $S_{II}\bar{h} = 0$. Since we have assumed Q is the identity matrix, from $S_I\bar{g} = 0$ we have $\alpha_g + \beta_g = 0$, which gives $(\alpha_g, \beta_g) = (1, -1)$. On the other hand, from $S_{II}\bar{h} = 0$ we obtain $\beta_h = 0$ which leads to $(\alpha_h, \beta_h) = (1, 0)$. Therefore, for the best locality in the first loop nest, array U should have diagonal memory layout whereas array V should be row-major.

Next we *propagate* these layout constraints to the second loop nest, and solve equation (8) for r_{12} and r_{22}. From

$$\begin{pmatrix} r_{22} & r_{12} & 0 & 0 \\ r_{12} & r_{22} & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 1 \\ 0 \end{pmatrix} = 0,$$

we have $r_{12} = r_{22} = 1$. Thus, a suitable transformation matrix is

$$R = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \text{ which gives } S = R^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}.$$

Using both loop transformation matrices and optimized memory layouts, the transformed program is shown in Figure 3(c). Notice that both loop nests exhibit good locality provided that array U is diagonally stored and array V is row-major. There is an additional transformation step which modifies this program for a language with fixed (canonical) memory layout; but since that step is almost mechanical, we omit it and refer the reader to [7] and [15]. The details of how to fill out a partially completed transformation matrix can be found in [12].

We stress that the second loop nest in Figure 3(b) cannot be optimized using pure loop (e.g., [12]) or pure data (e.g., [15]) transformations alone; because, there is spatial reuse in two orthogonal directions. This simple example shows that a combined approach might be useful for some programs.

4.1 Locality coefficient

To evaluate the amount of locality before and after the optimization process, we use a simple metric referred to as the *locality coefficient*. We define the locality coefficient of a loop nest as the number of references that exhibit locality (spatial or temporal) in the *innermost* loop. The locality coefficient of a series of loop nests is defined to be the sum of the locality coefficients of the individual nests. The locality coefficients of two different versions of a program can be used as a guide to decide which version is better than the other from the locality point of view. In the case of a tie, we favor the program with more temporal locality. Of course, this evaluation criterion for locality is very rough and assumes that all references have the same weight and the bounds of all innermost loops as well as the sizes of all arrays are of the same order. This model can be improved upon by taking into account a detailed profile information as well as the bounds of the arrays and the loops after a transformation; but, the exactness of the evaluation model is not very relevant for the purpose of this paper and the rest of the approach is independent of the particular locality evaluation criterion chosen. For example, the locality coefficient of the program shown in Figure 3(b) (assuming column-major layouts) is 1 whereas that of the optimized code in Figure 3(c) is 4 under optimal layouts.

4.2 Formulation for the general case

In the general case, when we handle a given loop nest during the global optimization process, some of the array layouts might be known, while the layouts of some arrays are yet to be determined. In such cases, we end up with a system of equations of the following type, which we call a *target system*:

$$S\bar{\xi} = \bar{0}. \qquad (10)$$

The systems given in equations (7) and (8) are two example target systems. Here S is a matrix which contains only last the column entries of the inverse of the loop transformation matrix, and $\bar{\xi}$ is a vector obtained from concatenating the hyperplane vectors. Our approach first brings this system into following form using elementary row interchange operations:

$$\begin{pmatrix} S_I & Z_I \\ Z_{II} & S_{II} \end{pmatrix} \begin{pmatrix} \bar{\xi}_k \\ \bar{\xi}_u \end{pmatrix} = \bar{0}, \qquad (11)$$

where S_I and S_{II} are non-zero sub-matrices and Z_I and Z_{II} are zero sub-matrices. It is easy to see that this is always possible. The vector $\bar{\xi}_k$ contains entries of hyperplane vectors (which correspond to memory layouts) that *have been* determined so far. The other vector, $\bar{\xi}_u$, consists of entries of the hyperplane vectors which *are to be* determined. After this point, our solution procedure comprises three steps:

(1) From $S_I\bar{\xi}_k = \bar{0}$, the entries of S_I are found;

(2) From the entries of S_I, the entries of S_{II} are determined; and

(3) From $S_{II}\bar{\xi}_u = \bar{0}$, the entries of $\bar{\xi}_u$ are solved.

We note that these three steps informally correspond to determining a loop transformation taking into account memory layouts obtained so far, and to determining memory layouts of (possibly a subset of) the remaining arrays whose layouts have not been determined so far. In the following, we explain these three steps in more detail.

Step (1) corresponds to solving a homogeneous system of equations. We first transform this system into $\bar{\xi}_k^T {S_I}^T = \bar{0}$, and then solve it for S_I. Of course, given a large number of references this system may not have a solution at all. In that case, we ignore some equations, and attempt to solve it again. The equations to be ignored should correspond to references that are least frequently accessed. The profile information might be useful in determining the access frequency of references.

In Step (2), the elements of S_{II} are determined from the elements of S_I found in the previous step. Although this step looks trivial, it is possible that an element which appears in S_{II} may not appear in S_I. In that case, we choose a value for this element arbitrarily avoiding picking up a zero value if all the other entries are zero.

Step (3) is very similar to Step (1), the only difference is that without taking the transposition, we start to solve the homogeneous system right away.

4.3 The most costly nest revisited

So far, we have assumed that the most costly loop nest will be optimized using data transformations alone. In this subsection, we first argue for this decision. Then we show how our approach can be made more powerful by considering different alternatives for the most costly nest.

Given a loop nest, determining both loop and data transformations together is not trivial as the problem requires in finding integer solutions to nonlinear systems of equations. If the search spaces for data and/or loop transformations are restricted, then an exhaustive search (although still costly) might be reasonable [3]. However, we insist on most general loop and data transformations. Having decided that we will not optimize the most costly nest with a combined (loop plus data) approach, the choice is between either pure loop or pure data transformations. In general, we prefer data transformations; because even if we choose loop transformations, we have to assume some fixed layouts for the arrays referenced. Moreover, for a single loop nest, data space transformations can be more successful than loop transformations since the latter is constrained by data dependences [7].

However, for some programs it might be the case that the best optimized program is the one in which the most costly nest is optimized using iteration space transformations alone. The reason is rather subtle. As mentioned previously, pure loop transformations can optimize temporal locality while pure data transformations cannot. If the most costly loop nest contains a number of references for which temporal reuse can be exploited in the innermost loop, then a pure loop based approach may result in a better code than a pure data based approach.

To solve this problem, our current approach is as follows. For the most costly nest, we consider two alternatives: pure loop and pure data transformations. Then we proceed for each version as explained in the previous sections, and finally come up with two different optimized program. Finally, we calculate and compare the locality coefficients of these two programs, and select the one with the larger coefficient. Notice that once the most costly nest is optimized, our approach will have some layout constraints for the remaining nests, and will proceed to optimize each nest using a combined approach which employs both loop and data transformations as explained.

Given the fact that the global locality optimization problem is NP-complete, and that in most programs the bulk of the execution time is spent in a couple of loop nests, we believe our approach is suitable for optimizing locality for multiple loop nests.

Table 1: Programs in our experiment set and different versions [`SIZE` in doubles; for `adi` and `transpose`, there is an outermost timing loop].

PROGRAM	COMMENT	SIZE
`btrix`	from Spec92	$5 \times 5 \times 175 \times 175$
`adi`	from Livermore	$3 \times 1024 \times 1024$
`transpose`	from NWChem (PNL) [14]	2048×2048

VERSION	COMMENT
`col`	fixed column-major memory layouts
`row`	fixed row-major memory layouts
`lopt`	loop-optimized version: no layout transforms
`dopt`	layout-optimized version: no loop transforms
`comb`	combined loop + data layout transforms

4.4 Impact on multiprocessor execution

Our locality algorithm strives to optimize locality in as many innermost loops as possible using a mix of loop and data transformations. This approach has can generate—as a byproduct—locality-free outer loops which are perfect candidates for parallelization. This is very desirable as otherwise parallelizing a loop which carries reuse is one of the main causes for inter-processor data sharing [12]. Intuitively, the more aggressive the compiler is in bringing the loops carrying reuse into innermost positions, the less false and true sharing will occur.

5 Performance results

In this section, we present our performance results to demonstrate the impact of our global locality optimization approach. We show execution times for three programs on an 8-node SGI Origin distributed-shared-memory machine. This machine uses 195MHz R10000 processors, 32KB L1 data cache and 4MB L2 unified cache. The C versions of the programs are compiled by the native C compiler using `-O2` option. The timings are taken using the *gettimeofday* routine. The programs that we use and the different versions that we experiment with are given in Table 1. The execution times in seconds for the programs in our experimental suite are given in Tables 2(a), 2(b) and 2(c) for `btrix`, `adi`, and `transpose`, respectively. The last rows in these tables give the percentage of improvement obtained using the `comb` version (our combined approach) over the next best time. It should be stated that in all versions after the transformations for locality the outermost loop which does not carry any data dependence has been parallelized. From these results, we conclude the following:

(1) Our combined approach is very successful in optimizing locality; to be specific, for all three programs, the `comb` version results in the best output code and the best execution time.

(2) The percentage of improvement depends on the relative performances of the other versions; therefore, there is not a single trend. It is also interesting to note that in two cases, `adi` and `transpose`, our approach (`comb`) generates the same code as `lopt` or `dopt`, respectively. This is due to our extended approach which considers both pure data- and pure loop-optimized versions of the most costly nest.

6 Related work

Recent years have witnessed a success in locality enhancing transformations. A majority of the techniques used are based on loop transformations. Wolf and Lam [18] describe reuse vectors and explain how they can be used for optimizing locality. Their method is sort of exhaustive and in some cases can only work with the approximate reuse vectors. Li [12] also considers reuse vectors but determines an appropriate loop transformation matrix in one go rather than resorting to an exhaustive search. Neither Li [12] nor Wolf and Lam [18] consider memory layout transformations; and since a loop transformation to improve locality of a reference can sometimes adversely affect the locality of another reference, both approaches may end up with unsatisfactory solutions. The cost of the methods mentioned is partly eliminated by a simple heuristic used by McKinley et al. [13]. Their method employs a simple cost formulation and considers loop permutation, loop reversal, loop fusion and distribution. In addition to having the disadvantages of an approach which is based on loop transformations alone, since they do not consider general non-singular loop transformations they may not be able to optimize some loop nests for which loop permutation does not work.

Considering the fact that linear loop transformations may be insufficient for some loop nests, some researchers have focussed on loop tiling [19] which in most cases can be accomplished via a combination of strip-mining and interchanging. The main question however is to select an appropriate tile size which is dependent on loop orders, array sizes, as well as cache sizes and degree of associativity [4].

More recently some researchers have considered data layout transformations which are simply restructuring of multi-dimensional arrays in memory. Leung and Zahorjan [11] present a technique which is based on non-singular data transformation matrices. They show that data transformations may be successful where loop transformations fail either because of conflicting requirements between different references to different arrays or simply because data dependences prevent the desired loop transformation. O'Boyle and Knijnenburg [15] also argue for data transformations. Apart from using it for optimizing spatial locality, they consider the use of data transformations for data alignment and page replication problems on parallel machines. There are two major problems with those techniques based on pure data transformations. First, data transformations cannot optimize for temporal locality which in turn may lead to poor register usage. Second, the impact of a layout transformation is global meaning that it affects (sometimes adversely) all the references to that array in all nests (assuming that no dynamic transformation is considered). Given large number of nests, it might be very difficult to come up with a data layout which satisfies as many nests as possible.

Yet another approach is to apply a combination of loop and data transformations for enhancing locality as we have done in this paper. Cierniak and Li [3] use this approach. Since they mainly focus on a single loop nest and the general problem exhibits non-linearity, they restrict search spaces for possible loop and data transformations, and resort to exhaustive search in this restricted search space. The data transformations they consider are permutations only; therefore, they cannot optimize banded matrix applications fully for which diagonal layouts are the most suitable. Kandemir et al. [8, 9] also consider data layout optimization techniques. Their techniques unify loop and data transformations in a unified framework, but restrict the data Even in the restricted space, they perform sort of exhaustive search.

In contrast to the previous work, our approach presented in this paper uses both loop and data transformations; and consequently can enjoy the advantages of the both. Also the search spaces that we consider for loop and data transformations are very general: For loop transformations we use general non-singular linear transformation matrices, and for memory layouts we can choose any optimal layout that can be expressed by hyperplanes. Finally, rather than limiting scope to a single loop nest we focus on a sequence of loop nests and propagate memory layouts across loop nests.

7 Conclusions

In this paper we have described a unified global approach for optimizing locality given a series of loop nests. During the optimization process, when a new loop nest is to be optimized, our approach first applies a loop transformation to it to satisfy the layout requirements for the references to arrays whose layouts have already been determined. It then determines suitable memory layouts for the remaining arrays referenced in the nest. The overall process is thus an alternating sequence of data (array layout) and loop (iteration space) transformations. Although the general problem looks difficult, we have shown in this paper that the whole process for a single nest can be formulated in a nice mathematical framework which is based on explicit memory layout representations. We have also shown that our approach looks more successful than existing locality enhancing (linear transformation) techniques whether they are pure loop based, pure data based, or a combination of both.

We are currently looking at the interaction between our framework and tiling, and plan to work on several problems such as evaluating relative performances of tiled code versus the resultant code from our approach and comparing our approach to a relatively new form of tiling, namely data-centric tiling [10]. In addition, we plan to investigate the effectiveness of *blocked* data layouts—in which the elements accessed by a tile are stored contiguously in memory—in improving the cache performance further. Also, we are working on extending our techniques for optimizing locality across program modules.

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 supported in part by NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422. P. Banerjee was supported in part by NSF under grant CCR-9526325 and in part by DARPA under contract DABT-63-97-C-0035.

References

[1] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th SIGPLAN*

Table 2: Execution times for benchmarks

(a): Execution times in seconds for `btrix`

version	number of processors							
	1	2	3	4	5	6	7	8
col	213.088	117.852	83.460	67.387	61.861	59.397	61.130	65.846
row	38.615	36.736	40.771	44.563	50.576	54.350	60.196	69.160
lopt	92.223	51.869	36.165	28.796	25.425	22.780	22.546	24.101
dopt	37.370	36.617	40.866	45.613	52.565	56.436	63.038	72.819
comb	33.669	21.533	19.013	17.568	18.962	18.731	18.002	19.361
% imprv.	10	41	47	39	25	18	20	20

(b): Execution times in seconds for `adi`

version	number of processors							
	1	2	3	4	5	6	7	8
col	72.102	52.589	39.539	36.853	29.730	30.361	26.202	28.402
row	18.528	12.432	10.718	10.216	8.901	8.531	7.774	8.513
lopt	14.842	7.975	5.435	4.290	3.581	3.129	2.750	2.877
dopt	15.264	10.393	8.807	9.859	7.459	7.642	7.270	6.808
comb	14.842	7.975	5.435	4.290	3.581	3.129	2.750	2.877
% imprv.	0	0	0	0	0	0	0	0

(c): Execution times in seconds for `transpose`

version	number of processors							
	1	2	3	4	5	6	7	8
col	73.342	38.159	27.114	21.328	18.190	16.910	14.677	14.843
row	78.483	40.653	28.757	22.472	19.165	16.929	15.389	14.899
lopt	73.342	38.159	27.114	21.328	18.190	16.910	14.677	14.843
dopt	30.244	16.609	11.559	8.863	7.389	6.415	5.633	6.674
comb	30.244	16.609	11.559	8.863	7.389	6.415	5.633	6.674
% imprv.	0	0	0	0	0	0	0	0

Symp. Prin. & Prac. Par. Prog., 1995.

[2] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. SIGPLAN Conf. Prog. Lang. Design & Impl.*, pp. 112–125, 1993.

[3] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. *Proc. SIGPLAN Conf. Prog. Lang. Design & Impl.*, 1995.

[4] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proc. SIGPLAN Conf. Prog. Lang. Design & Impl.*, 1995.

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

[6] C. Huang and P. Sadayappan. Communication-free partitioning of nested loops. *Jou. Par. & Dist. Comp.*, 19:90–102, 1993.

[7] 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.

[8] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proc. 11th ACM International Conference on Supercomputing*, pp. 269–276, 1997.

[9] 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. Para. Arch. & Comp. Tech. (PACT 97)*, pp. 236–247.

[10] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Prog. Lang. Design & Impl.*, 1997.

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

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

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

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

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

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

[17] E. Torrie, C-W. Tseng, M. Martonosi, and M. W. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proc. 1995 Int. Conf. Para. Arch. & Comp. Tech. (PACT 95).*

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

[19] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proc. 3rd SIAM Conference on Parallel Processing for Scientific Computing*, pp. 357–361, 1987.

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