Appears in Journal of Parallel and Distributed Computing, September 1999

A Matrix-Based Approach to Global Locality Optimization*

Mahmut Kandemir† Alok Choudhary ‡ J. Ramanujam§ Prith Banerjee ‡

Abstract

Global locality optimization is a technique for improving the cache performance of a sequence of loop nests through a combination of loop and data layout transformations. Pure loop transformations are restricted by data dependences and may not be very successful in optimizing imperfectly nested loops and explicitly parallelized programs. Although pure data transformations are not constrained by data dependences, the impact of a data transformation on an array might be program-wide; that is, it can affect all the references to that array in all the loop nests. Therefore, in this paper we argue for an integrated approach that employs both loop and data transformations. The method enjoys the advantages of most of the previous techniques for enhancing locality and is efficient. In our approach, the loop nests in a program are processed one by one and the data layout constraints obtained from one nest are propagated for the optimizing the remaining loop nests. We show a simple and effective matrix-based framework to implement this process. The search space that we consider for possible loop transformations can be represented by general non-singular linear transformation matrices and the data layouts that we consider are those that can be expressed using hyperplanes. Experiments with several floating-point programs on an 8-processor SGI Origin 2000 distributed-shared-memory machine demonstrate the efficacy of our approach.

Keywords: data reuse, locality, memory hierarchy, parallelism, loop transformations, array restructuring, data transformations.

Corresponding Author:

Prof. Prith Banerjee
Department of Electrical and Computer Engineering
Northwestern University
L 497, Technological Institute
2145 Sheridan Road
Evanston, IL 60208-3118
Phone: (847) 491-4118, Fax: (847) 467-4144
Email: banerjee@ece.nwu.edu

*A preliminary version of this paper appears in the proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98).

†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

1 Introduction

The processing power of high performance computers continues to increase dramatically. In the mean time, the rate of increase of speed of the memory subsystems has not kept pace with the rate of improvement in processor speeds [23]. Therefore, 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. Even with the use of memory hierarchy, realizing the highest levels of performance on current machines requires careful orchestration of programs. Fortunately, optimizing compilers have been instrumental in relieving some of the burden on the users; and they play an increasingly critical role in the face of the rapid evolution of architectures.

A recent study shows that a group of highly parallelized scientific benchmarks spend as much as a quarter to a half of their execution times waiting for data from memory [55]. It is now widely accepted that 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 have been incorporated into a single framework using a matrix representation of these transformations [59]. Among the techniques used are unimodular [7] and non-unimodular [39] iteration space transformations as well as tiling [58, 56, 37]. These techniques share the following two characteristics:

(1) they attempt to improve 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.

However, loop transformations have some important drawbacks. First, they may not always be legal due to dependence constraints [59]. Second, it might be difficult to find a loop transformation that improves the locality of *all* the arrays referenced in the nest. Third, loop transformations are not very successful in optimizing locality for imperfectly nested loops; for example, Kodukula and Pingali [34] show that handling imperfectly nested loops within a loop transformation framework requires a somewhat different approach to the problem. And finally, as noted by Cierniak and Li [15], loop transformations are not very successful in optimizing explicitly parallelized programs since these programs include parallel execution specifications (e.g., annotations) as well as synchronization constructs; this renders locality optimization very difficult.

Proper layout of data in memory may also have a significant impact on the performance of scientific computations on multiprocessor machines. In fact, as shown by O'Boyle and Knijnenburg [44], Leung and Zahorjan [38] and Kandemir et al. [27], techniques that decide good data layouts (e.g., row-major or column-major storage of large arrays) can improve the performance of dense array codes significantly. We refer to the optimization of data layouts as *data transformations.* Unlike loop transformations, data transformations are not constrained by data dependences, and are easily applicable to imperfectly nested loops and explicitly parallelized programs. However, there are two important drawbacks associated with data transformations. First, they cannot optimize temporal locality. As a result of this, pure data transformations can generate programs that do not fully utilize the registers in the underlying architecture. But, more importantly, the impact of a data transformation is *global*, i.e., the effect goes beyond a single loop nest boundary. For example, transforming the memory layout of a multi-dimensional array from column-major to row-major to improve the locality of a reference to the said array in one loop nest can adversely affect the locality of the other references to the same array in other loop nests.

Therefore, it is reasonable to expect that a *unified* (integrated) approach that employs both loop and data transformations should, in principle, perform better than approaches based on *pure* loop or *pure* data transformations. In this paper, we explore this possibility. Specifically, we seek an answer to the following question:

"Given a program that consists of a number of loop nests, what are the appropriate loop transformations for the loop nests and the appropriate data transformations for the multi-dimensional arrays referenced such that the overall cache locality will be very good?"

Our goal is to derive a unified framework that satisfies the following requirements:

- The framework should be fast; that is, it should not resort to exhaustive search to find memory layouts and/or loop transformations. Some of the previous work in this area [15, 30, 31] use techniques that combine some restricted sets of loop and data transformations; but they use a kind of exhaustive search to achieve their goal. Since no quantification of the compilation times is available, it remains unclear whether such techniques are fast and effective in practice.

- The framework should be fairly general. It should be able to use (when necessary) the general non-singular loop transformations. Previous work based on pure loop transformations such as those of Wolf and Lam [56] and McKinley et al. [43] use unimodular loop transformations which may not be sufficient for optimizing locality in loop nests that require non-unimodular transformations. The framework should also use general data transformations. Works published in [15], [30] and [4] consider only a limited subset of data transformations. For example, they do not consider diagonal (skewed) data layouts which can prove very useful for banded-matrix applications.

- The framework should be able to optimize both temporal and spatial locality. This is a reasonable requirement as we use both loop and data transformations; so, in principle, we should be able to optimize for both types of locality. Note that locality enhancing techniques based on pure data transformations (e.g., [44], [38], [27]) cannot optimize temporal locality.

- The framework should be able to optimize imperfectly nested loops as well as explicitly parallelized programs. Since we use data transformations as a part of our unified framework, we should be able to optimize those important cases where most of the techniques based on pure loop transformations fail.

In this paper we present a unified framework that attempts to satisfy the goals set above. McKinley et al. [43] suggests the following three-step high-level approach for locality optimization:

(1) Optimize temporal and spatial locality through loop transformations such as loop permutation, reversal, fusion, distribution, and skewing;

(2) Optimize cache locality further using tiling [58] which is a combination of strip-mining and loop permutations;

(3) Optimize register usage through unroll-and-jam and scalar replacement [11, 12].

Our approach can be seen as an attempt to increase the effectiveness of the first step, and is orthogonal to the other two. We recommend applying tiling and register level optimizations following our unified approach to achieve the best results. In fact, as will be discussed later in the paper, our approach helps to obtain better results from the second and the third steps. 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 can model the data transformations using vectors and 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.

The rest of the paper is organized as follows. Section 2 presents the motivation for attacking the global locality optimization problem and describes our approach informally. Section 3 describes the fundamental concepts used in our approach. Section 4 presents the details of our approach through several examples. In that section we first

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

do i = 2, N                  do i = 1, N          do i = 1, N
  do j = 1, N-1                do j = 1, N          do j = 1, N
    W(i,j) = W(i-1,j+1)          U(j,i) = ···         V(i,j) = U(j,i)
  end do                       end do               end do
end do                       end do               end do
        (a)                       (b)                  (c)
```

Figure 1: Three program fragments: **(a)** Loop transformations fail, data transformations work. **(b)** Data transformations fail, loop transformations work. **(c)** Integrated loop and data transformations are necessary.

focus on a general technique for improving locality on both uniprocessors and multiprocessors, and then discuss a number of multiprocessor-specific issues. Section 5 presents experimental results obtained on an eight node SGI Origin 2000 distributed-shared-memory multiprocessor. Section 6 reviews related work on locality enhancing techniques and finally Section 7 summarizes the paper with an outline of ongoing and future work.

2 Our approach

2.1 Motivation

As noted earlier, there is a vast amount of work in optimizing locality for a single loop nest using *iteration space* (loop) transformations [58, 56, 39, 43]. Recently some authors [4, 44, 38] have proposed techniques to optimize spatial locality in a loop nest using *data space* (array layout) 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 the references to that array in all the loop nests in the program. We show in this paper that such a global impact can in fact be exploited using a proper mix of data and iteration space transformations.

In order to motivate the discussion, we consider the program fragments shown in Figure 1. The first loop nest shown in Figure 1(a) accesses two arrays. It can be shown that it is not possible to optimize spatial locality (by ensuring stride one accesses in the inner loop) for both references using pure loop transformations. The reason is that the innermost loop accesses the two arrays in a different manner. Assuming a column-major memory layout for both the arrays, the locality for array *V* is good whereas that of array *U* is not. A simple loop interchange transformation [59] would improve the locality for array *U*, but would degrade the locality of *V*. However, without any loop transformation, using a column-major memory layout for array *V* and a row-major memory layout for array *U* will result in good locality for both the references. If the default memory layout is column-major, this solution will involve a data transformation for array *U* (from column-major to row-major). The second loop nest in Figure 1(a) illustrates another problem with pure loop transformations. Again, assuming a column-major memory layout, both references to array *W* exhibit poor locality as the inner *j* loop skips through the columns. Unfortunately, the obvious solution of interchanging two loops is not *legal* here due to data dependences [59]. One simple solution is again a data transformation for array *W*; that is, transforming its memory layout from column-major to row-major.

Now consider the program fragment shown in Figure 1(b). In this fragment there are two loop nests accessing the same array in different fashions. The techniques based on pure data transformations cannot do much for this program fragment. The reason is that no matter what the layout of array U is, the locality in one of the loop nests will be poor unless the loops are transformed. It is easy to see that without any data transformation (assuming a column-major memory layout) just interchanging the loops in the first nest (provided that it is legal to do so) will solve the problem.

Figures 1(a) and (b) show the cases where pure data transformations and pure loop transformations, respectively, are required and sufficient. Figure 1(c) illustrates an example for which neither pure loop nor pure data transformations work. In that case, one solution is to select row-major layout for array U and column-major layout for array V, and interchange the loops in the second nest. Notice that such a solution requires the application of *both* loop and data transformations. These three program fragments show us that neither loop nor data transformations subsume the other and sometimes a unified (integrated) approach that uses a mix of loop and data transformations might be necessary.

2.2 Overview

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 [59]; in this canonical form, a program contains two types of references, those that occur inside loop nests and those that occur between loop nests. In our approach, only the references within loop nests can have an effect on 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 [5, 3]. This is a bipartite graph that 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. We need to be careful in defining the cost of a loop nest. Although it may sound logical to profile the code and use the uniprocessor execution times as the costs of the loop nests, it is not a good idea. The reason is that execution times are intricately associated with memory layouts (which will be determined by our approach). Therefore, what we need is a layout-independent cost criterion. In the following section, we discuss a cost criterion based on the *weights* of the references in a program. Notice that the ordering of loop nests according to a cost criterion is *not* a modification or transformation of the structure of the program; but rather a step to determine the order in which the loop nests will be considered (optimized) by our locality optimization 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* to the remaining nests 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 important cases:

Case 1**:** There are no memory layout constraints; that is, none of the arrays have a determined memory layout so far, and we have complete freedom in choosing loop transformation (except as constrained by data dependences in the loop nest under consideration). 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. In general, this is the case for the most costly nest in a connected component.

Case 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. This is the most commonly encountered case.

Case 3**:** We have restrictions on the iteration space other than those due to data dependences. These restrictions, called *additional constraints* (as against layout constraints), are in general related to parallelism; typically, these restrictions can arise due to the need to reduce (or perhaps eliminate) false sharing [25, 54] or the need to exploit the largest granularity of parallelism [7]. The goal is to determine the memory layouts for the arrays referenced.

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 arise.

After all the layouts and loop transformations are found, the compiler applies these transformations and generates the optimized code. In the next section we give details of the main elements of our approach.

3 Elements and scope of our approach

3.1 Canonical form

We assume that the only control flow that we have is loop structures. We simply treat *if*–constructs as if both branches always execute. In the canonical form, a program consists of just 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. [43], which uses loop fusion and loop distribution. Loop fusion [59] takes adjacent loop nests and combines their bodies into a single body collapsing their iteration spaces into a single iteration space. It can create perfectly nested loops from imperfectly nested loops, and can improve cache locality as well as register usage. Loop distribution [59], on the other hand, creates multiple loop nests from a single loop nest by separating the independent statements in a single loop into multiple loops. It is typically used to break the inter-statement dependence cycles in loop nests with multiple statements to enable parallelism.

Our algorithm first normalizes [59] each nest such that the step size of each loop becomes one. It then considers imperfectly nested loops and transforms them into perfectly nest loops using a combination of loop fusion, loop distribution and code sinking (when necessary). 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 [43]; a sketch of the algorithm Canonical is shown in Figure 2. The details of the functions fuse, distribute, and code-sink are omitted for lack of space and not relevant for the purpose of exposition. Notice that, for the sake of clarity, the figure does not show the final loop fusion pass. Also note that like Wolf et al. [57] we favor fusion over distribution largely because fusion improves cache reuse and results in less overhead.

As an illustration of the working of Canonical, we consider the program fragment shown on the left part of Figure 3. 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. As a result of these transformations, the initial program is converted to a code that consists of three perfectly nested loop nests.

3.2 Interference graph

Next the compiler builds an interference graph similar to that used by Anderson and Lam [5] in solving the automatic data distribution problem for parallel machines. This is a bipartite graph (V_n, V_a, E) where V_n is the set of loop nests,

Canonical (Δ)
INPUT: *A set of loop nests* $\Delta = \{\delta_1, \delta_2, \cdots\}$
OUTPUT: *A set of loop nests* $\Delta' = \{\delta'_1, \delta'_2, \cdots\}$
  foreach $\delta_i \in \Delta$
    if *(δ_i is perfectly-nested)* continue
      else
        if *(δ_i has (non-loop) statements sandwiched between loop nests)* code-sink(*statements*)
        endif
        let $\{\sigma_1, \sigma_2, \cdots, \sigma_r\}$ *be the resulting loop nests in* δ_i *after the code-sinking*
        Canonical*($\{\sigma_1, \sigma_2, \cdots, \sigma_r\}$) /* recursive call! */*
        let $\{\sigma'_1, \sigma'_2, \cdots, \sigma'_t\}$ *be the resulting loop nests in* δ_i *after the recursive call*
        if *(fusion is legal and profitable)* fuse*($\{\sigma'_1, \sigma'_2, \cdots, \sigma'_t\}$)* endif
          else if *(distribution is legal)* distribute δ_i *over* $\{\sigma'_1, \sigma'_2, \cdots, \sigma'_t\}$ endif
    endif
  endforeach

Figure 2: Overview of Canonical – an algorithm to transform a program to our desired form.

V_a is the set of arrays, and E is the set of edges between loop vertices and array vertices. There is an edge $e \in E$ 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 3, we have two connected components. Each connected component is fed into our global locality optimization algorithm explained in the rest of this paper; that is, our algorithm works on a single connected component at a time.

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. 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 $(\jmath_1, \jmath_2, \cdots, \jmath_m)$ that satisfy the relation

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

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* [24, 47]. 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 will be omitted. A *hyperplane family* is a set of hyperplanes with the same coefficients but with a different constant (c value).

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 the hyperplane equation is $\jmath_2 = c$ where $1 \leq c \leq N$. For example, $\jmath_2 = 5$ represents the fifth column of the array. An important fact about the hyperplanes is that two array elements $\bar{J}$ and $\bar{J}'$ belong to the same hyperplane $\bar{g}$ if

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

Notice that the multiplications in this equation are dot products and that all the transpose symbols are omitted. Returning to our two-dimensional column-major array, since the array elements $(1, 5)$ and $(4, 5)$ satisfy Equation (2) for

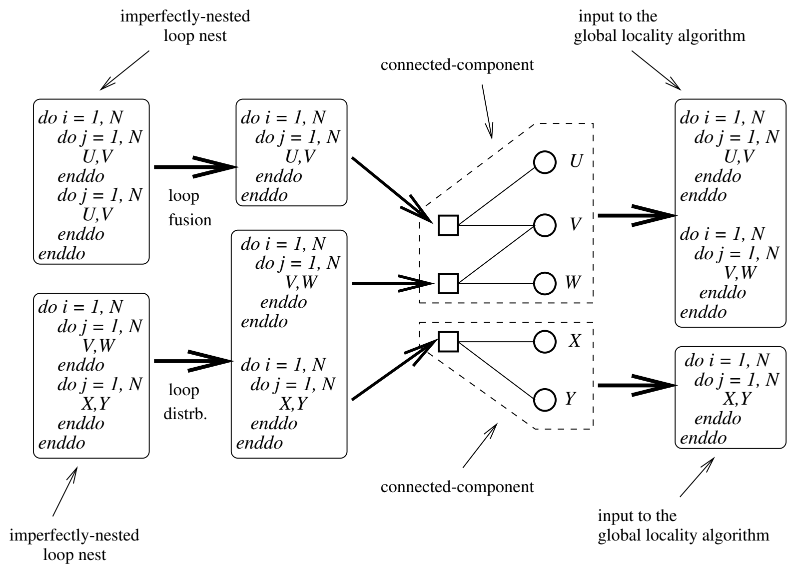

Figure 3: Example application of locality optimization algorithm.

$\bar{g} = (0, 1)$, they belong to the same hyperplane that can be identified with $c = 5$. 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 (for a two-dimensional array) 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 contiguously in memory. Similarly, the hyperplane vectors given by $(1, 0)$ and $(1, 1)$ correspond to two-dimensional row-major and anti-diagonal memory layouts, respectively. Figure 4 shows a few possible memory layouts for a two-dimensional 8×8 array and the associated hyperplane vectors below them. 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 [27]. 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 the set of layouts given in Figure 4 is not exhaustive, as, for instance, the hyperplane vector $(3, -4)$ also represents a possible memory layout.

Our representation as explained so far has two problems. First, although, say, a column-major layout (as defined in Fortran) totally defines the relative order of columns as well with respect to each other, our representation does not necessarily specify such a relative order among the hyperplanes. Although, the hyperplane coefficients (c values) can be used to order the hyperplanes, in programs where array sizes far exceed size of the available cache memory, the effect of the relative order of hyperplanes is of secondary importance. The second problem is related to our definition of spatial locality which is coarse-grained and different from those of the previous work (e.g., [56]). 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. We believe that this is also not a significant issue. In fact, our technique works as if it is operating on an array space where the boundaries are lifted.

For two-dimensional arrays, a single hyperplane family is sufficient to partially 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

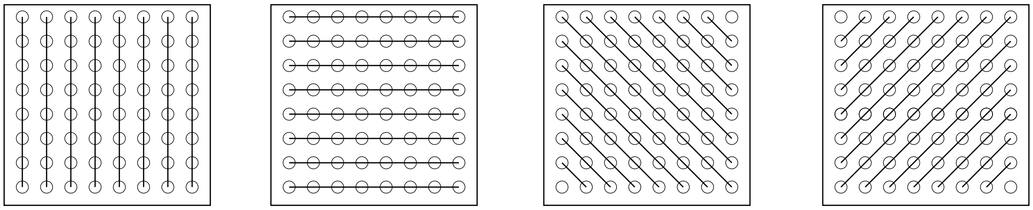

Figure 4: Example memory layouts for two-dimensional arrays and their hyperplane vectors.

$\bar{h} = (0, 1, 0)$. We can write these two hyperplanes collectively as a *layout constraint matrix* or simply a *layout matrix* [27]

$$\mathcal{C}_U = \begin{pmatrix} \bar{g} \\ \bar{h} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

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

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

The elements that exhibit spatial locality should be stored in consecutive memory locations. Notice that this representation of column-major layout of a three-dimensional array matches naturally with the column-major layout concept of Fortran; because, in Fortran, in order for two array elements $\bar{J}$ and $\bar{J}'$ to map on the same column all array indices except possibly the first one (column index) should be equal. We note that the Equations (3) and (4), together, enforce the mentioned (equality) conditions. This idea of representing memory layouts using layout matrices can be easily generalized to higher dimensions; the details are beyond the scope of this paper and can be found elsewhere [27].

3.4 Optimizing a loop nest using pure loop or pure data transformations

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 very important for parallel architectures with some form of cache hierarchy. To understand the effect of a loop transformation let us represent a loop nest of depth n that 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 loop-index-independent variables. We can model each array reference using an *access matrix* $\mathcal{L}$ and an *offset* (*constant*) *vector* $\bar{o}$ [56, 39]. 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 $m \times n$

```
do i = 3, N
 do j = 3, N
  U(i,j)=(V(i-2,j)+V(i,j)+V(i,j-2))/3
 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 5: **(a)** An example loop nest. **(b)** Original fragment. **(c)** Optimized version of (b).

and the offset vector is m-dimensional.

The class of iteration space transformations we are interested in can be represented using non-singular square *transformation matrices* [59]. For a loop nest of depth n, the iteration space transformation 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 $\mathcal{L}T^{-1}\bar{I}' + \bar{o}$ meaning that the new access matrix is $\mathcal{L}T^{-1}$. The problem investigated in works such as [56] and [39] 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 code shown in Figure 5(a). Assuming column-major memory layouts for arrays 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 touch 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 [39] optimizes this nest by interchanging the loops, which is legal here. This loop transformation can be represented by a unimodular 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 $\bar{I}_{next} = (i, j+1)$ be two consecutive iterations. We focus on array U, and a similar analysis 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 in the innermost loop, the constraint $\bar{g}(\mathcal{L}\bar{I} + \bar{o}) = \bar{g}(\mathcal{L}\bar{I}_{next} + \bar{o})$ should be satisfied, where $\bar{g}$ represents an optimal (desired) 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 in the innermost loop. Since, in our example, $\mathcal{L}$ is the identity matrix, we have $\bar{\ell} = (0, 1)^T$. Choosing $\bar{g}$ from the null space of $\bar{\ell}$ gives us $\bar{g} = (1, 0)$, which, as mentioned earlier (see Figure 4), corresponds to a row-major memory layout. The details of selecting a suitable vector from the Ker set are not important for the purposes of this paper and are fully explained in [27]. To sum up, in order to have a good spatial locality in the innermost loop, we have to change (transform) the memory layout of U (and that of V) from column-major to row-major.

Of course, *determining* the layout for a given array is just one part of the story; the other part is *implementing* this layout in a given compiler that uses a single *canonical layout* (i.e., default, base layout) for all the arrays (e.g., column-major in Fortran). Since the optimal memory layouts will be implemented in a compiler that uses a fixed default layout for all arrays, suitable data transformations should be applied for the references to the arrays whose optimal layouts are different from the default layout. Fortunately, this process is rather mechanical. Assuming $\mathcal{C}_{def}$ is the layout matrix representing the default layout (e.g., column-major in Fortran), in our framework, an optimal memory layout (denoted by the matrix $\mathcal{C}_{opt}$) for an array can be implemented in three steps:

(1) From $\mathcal{C}_{def}M = \mathcal{C}_{opt}$, a suitable *data transformation matrix* M of size $m \times m$ for an m-dimensional array is derived;

(2) The subscript expression of each reference to the array is transformed using M as a linear transformation matrix: $\mathcal{L}\bar{I} + \bar{o} \mapsto M\mathcal{L}\bar{I} + M\bar{o}$;

(3) The array bounds are also transformed accordingly. For example, if a two-dimensional array is of size $N \times M$ and is column-major, when the layout is converted to row-major it will be of size $M \times N$.

This implementation framework is from [27] and the second and the third steps are similar to the techniques proposed by Leung and Zahorjan [38] and O'Boyle and Knijnenburg [44]. The first step, however, allows us to use any type of layout as the base layout and transform the references in question such that the program will get executed under the base layout (represented by L_{def}) and will achieve good locality. We will not explicitly focus on these steps in this paper, largely because they are straightforward. We refer the interested reader to [27] and [44] for an in-depth discussion of issues related to code generation after a data transformation.

3.5 Ordering loop nests

We order the loop nests for processing by our locality optimization algorithm according to a *cost criterion*. Our cost criterion is based on the concept of the *weight* of a reference, defined as the number of times the reference is accessed at run-time. The unknown loop bounds and array sizes, and conditionally executed constructs make it very hard sometimes to calculate the exact weight of a given reference; currently we use profile information to get the average values of loop bounds and array sizes as well as to estimate conditional probabilities if they happen to be unknown at compile time.

Suppose that there are s loop nests in the program and r different arrays. Let $R_{\alpha\beta\gamma}$ be the γ^{th} reference to array β ($1 \le \beta \le r$) in loop nest α where $1 \le \alpha \le s$. Further assume that the loops are normalized and $t(l)$ returns the trip count (the number of iterations) of a loop l in a given nest. Then we can define the weight (cost) of a reference $R_{\alpha\beta\gamma}$ as

$$CR(R_{\alpha\beta\gamma}) = \prod_{l \text{ encloses } R_{\alpha\beta\gamma}} t(l).$$

Note that it is possible that not all of the loops in the nest enclose the reference in question. The weight (cost) of a loop nest α, on the other hand, is

$$CN(\alpha) = \sum_{R_{\alpha\beta\gamma} \text{ appears in } \alpha} CR(R_{\alpha\beta\gamma}).$$

Notice that the function CN can be used to order the nests; that is, a loop nest α will be processed (optimized) before a loop nest α' if and only if $CN(\alpha) > CN(\alpha')$. A good thing about this function is that it is *independent* of the memory layouts and roughly indicates the importance of a given loop nest with respect to the others. If needed, conditional probabilities can also be taken into account. Assuming that a reference $R_{\alpha\beta\gamma}$ will be accessed with a probability of $p_{\alpha\beta\gamma}$ where $0 \le p_{\alpha\beta\gamma} \le 1$, we can define the weight of a reference $R_{\alpha\beta\gamma}$ as

$$CR(R_{\alpha\beta\gamma}) = p_{\alpha\beta\gamma} \prod_{l \text{ encloses } R_{\alpha\beta\gamma}} t(l).$$

3.6 Scope of our work, extensions, and limitations

Our work focuses on deriving appropriate combinations of loop and data transformations for optimizing locality. The scope of our work is dense regular (affine access) matrix codes whose access patterns can be detected at compile-time. However, Leung and Zahorjan [38] show that data transformations can also be used to handle a *restricted* set of non-affine array access patterns. We stress that the set of irregular access patterns optimized by the mentioned technical report can also be handled by our approach. But, in this paper, we exclusively focus on affine array accesses.

It should also be noted that data transformations can be used for eliminating conflict misses and improving cache line (or page) alignment as well. Padding [6, 48], a special kind of data transformation for eliminating conflict misses, is orthogonal to our approach and, when necessary, can be used after the layouts have been transformed.

The shift-type of data transformations [44] are also not considered in this paper; however, our approach can easily be extended to include the shift-type of transformations as follows. We can think of a data transformation as a pair $(M, \bar{\eta})$ where M is an $m \times m$ data transformation matrix as before and $\bar{\eta}$ is an m-dimensional shift vector. Then the problem is to determine M as well as $\bar{\eta}$. A simple way of doing this is first determining M and then taking the cache line (or page) size into account to find an appropriate value for $\bar{\eta}$.

Although not considered here, data transformations can also be applied to one-dimensional arrays as well provided that they can be *de-linearized* first [16]. We are currently working on a *cache conscious* de-linearization scheme based on Maslov's algorithm [41].

We should also mention that our approach does not use any symbolic analysis; rather, it relies on profiling (when necessary) to obtain estimations about array sizes, loop bounds, conditional probabilities and so forth. We acknowledge, however, that locality optimizations (like parallelism optimizations) can benefit a lot from information provided by symbolic analysis techniques [10, 9, 22]. For example, using symbolic analysis, the compiler might be able to judge better whether applying tiling will be beneficial. In particular, if the compiler can detect that the entire data used by a loop nest does not cause cache overflow, then tiling is unnecessary.

Finally, there are four other important issues that need to be addressed. First, although data transformations are not constrained by data dependences, their applicability is restricted by the parameter passing rules and the sequence association rules of the language in question. In some cases, it might be necessary to apply run-time checks to determine if it is safe to apply a candidate data transformation. We do not investigate this issue in this paper and assume that the data transformations we apply are always legal. Of course, this may not always be true; in those cases techniques proposed by Chandra et al. [13] can be used.

The second important issue is the propagation of layout transformations across procedure boundaries. Currently, the scope of our work is limited to one procedure at a time and the experimental results presented later on are obtained on inlined [2] codes. We are working on a framework for propagating array layouts across procedures for the restricted case where there is no array re-shaping (and using de-linearization when necessary). The approach is similar in spirit to the solution proposed by Anderson [3] for data distribution problem and is based on a bottom-up and a top-down traversal of the call graph [2] of the program being analyzed. In those cases where arrays are re-shaped across procedures, we may need to apply explicit data transformations at run-time. Recently, O'Boyle and Knijnenburg [45] have proposed new techniques to propagate data transformations in the presence of re-shaping between procedure boundaries. When necessary, our framework can be modified to include the techniques proposed in [45]; we postpone a complete treatment of cache locality optimization in an inter-procedural setting to a future work.

Thirdly, our loop and data transformations do *not* cover the entire space of possible transformations. Our loop transformations can be represented by square non-singular loop transformation matrices [39], i.e., they are *linear*. For the best results, they need to be combined with the non-linear loop transformations such as tiling, loop fusion and distribution [59]. In this paper we briefly investigate the interaction with tiling; however we do not study the best ways of combining linear and non-linear loop transformations. Our data transformations, on the other hand, can be represented using hyperplanes. Therefore, they should cover all data transformations that can be represented by non-singular data transformation matrices [44]. Note that we do not handle blocked layouts [3]; the question of whether blocked layouts can be represented—at least in a constrained form—by hyperplanes merits study.

And finally, our solution procedure explained in the next section handles loop nests one by one, and fixes some memory layouts in each loop nest and propagates these layouts to the remaining nests. While the experiments indicate that our approach is quite successful, an alternative approach that takes a more *global view* is also possible. In such an approach, the arrays and the nests are represented using an interference graph as explained in Section 3.2. In the interference graph, each edge is replaced by a bidirectional arcs, and moving from one vertex (array vertex or nest vertex) to another is interpreted as using the solution of the former in solving the latter. Then, the problem of optimizing locality for the maximum number of references can be rephrased as finding a maximum-branching solution that satisfies as many edges possible. A similar approach has been used by Dion et al. [18] for solving the optimal alignment problem for distributed-memory message-passing machines.

4 Unified loop and data 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 integrate these two optimization techniques in a unified framework. 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 (see Section 4.5). We define $\bar{I}' = (i, j)$ and $\bar{I}'_{next} = (i, j+1)$ as two consecutive iteration vectors *after* the transformation. Assume that we use a 2×2 non-singular loop transformation matrix T and let $Q = T^{-1}$ be the inverse of the transformation matrix. Further assume that $\bar{g}$ represents the desired memory layout. After the transformation, in order to have spatial locality in the innermost loop (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

$$\bar{g}\mathcal{L}\bar{q} \quad = \quad 0, \tag{5}$$

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 (5) will be satisfied.

Notice that this equation is *non-linear* and it is not trivial to solve. However, if either $\bar{g}$ or $\bar{q}$ is known, then it is easy to determine the other by solving an homogeneous system. In this paper, to start the solution process, we first fix either $\bar{g}$ or $\bar{q}$ for the first nest. For the remaining nests, since the layouts involved would be fixed during solution of the previous nest(s), we can determine the loop transformation matrix without much difficulty.

Note also that Equation (5) is with regard to a single loop nest and a single reference. In order to optimize locality globally (i.e., procedure-wide), we should set up and solve simultaneously the equations similar to Equation (5) for every reference in every loop nest. Of course, given a large number of loop nests and references, this system of equations may have only a *trivial* solution (i.e., zero vectors for $\bar{q}$ or $\bar{g}$), in which case we need to ignore some equations. In our current approach, we use profile information to decide the equations to be ignored.

Notice also that Equation (5) can be easily modified to handle temporal locality as well. In order to have temporal locality in the innermost loop, Equation (5) should be satisfied no matter what the layout of the array in question is. In mathematical terms, we should have

$$\mathcal{L}\bar{q} \quad = \quad 0. \tag{6}$$

The interpretation of this last equality is that the last column of the inverse of the loop transformation matrix should belong to the null set of *all the rows* of the access matrix.

We illustrate the process using an example first. Consider the program fragment given in Figure 5(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 let $Q = T^{-1}$ and $R = S^{-1}$. Also let

$$\bar{q} = \begin{pmatrix} q_{12} \\ q_{22} \end{pmatrix} \text{ and } \bar{r} = \begin{pmatrix} r_{12} \\ r_{22} \end{pmatrix}$$

be the last columns of Q and R, respectively. Finally, let $\bar{g} = (\alpha_g, \beta_g)$ and $\bar{h} = (\alpha_h, \beta_h)$ represent the optimal layouts of U and V, respectively. Using Equation (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:

$$(\alpha_h, \beta_h) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} q_{12} \\ q_{22} \end{pmatrix} = 0.$$

Similarly, for the references to array U in the second nest, we have

$$(\alpha_g, \beta_g) \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_{12} \\ r_{22} \end{pmatrix} = 0 \text{ and } (\alpha_g, \beta_g) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} r_{12} \\ r_{22} \end{pmatrix} = 0.$$

We can write these equations collectively as

$$\begin{pmatrix} q_{12} + q_{22} & q_{22} & 0 & 0 \\ 0 & 0 & q_{12} & q_{22} \\ r_{22} & r_{12} & 0 & 0 \\ r_{12} & r_{22} & 0 & 0 \end{pmatrix} \begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \end{pmatrix} \quad = \quad \bar{0}. \tag{7}$$

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 the last columns of the inverses) as well as optimized memory layouts (i.e., their representative hyperplane vectors). 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 (7) is very difficult. What we need is a heuristic that works fast in practice and generates acceptable near-optimal solutions. In the following we show how to solve such a system by fixing some unknowns at specific values. We note that (7) can be divided into two sub-matrix equations, each for a single nest:

For the first nest: $$\begin{pmatrix} q_{12}+q_{22} & q_{22} & 0 & 0 \\ 0 & 0 & q_{12} & q_{22} \end{pmatrix} \begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \end{pmatrix} = \bar{0}, \qquad (8)$$

For the second nest: $$\begin{pmatrix} r_{22} & r_{12} & 0 & 0 \\ r_{12} & r_{22} & 0 & 0 \end{pmatrix} \begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \end{pmatrix} = 0. \qquad (9)$$

Notice that there is a *coupling* between (8) and (9) due to the layout (hyperplane) vectors. Let us first focus on (8), 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 will discuss this decision in detail. We can now think of (8) in a block form as shown below:

$$\left(\begin{array}{cc:cc} q_{12}+q_{22} & q_{22} & 0 & 0 \\ \hdashline 0 & 0 & q_{12} & q_{22} \end{array}\right) \begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \end{pmatrix} = \bar{0} \qquad (10)$$

This last equation can be written symbolically as follows (where S_I and S_{II} corresponds to non-zero sub-matrices in (10)):

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

Essentially, now we need to solve two equations: $S_I\bar{g} = 0$ and $S_{II}\bar{h} = 0$. Since we have assumed that 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 (see Figure 4).
Next we *propagate* these layout constraints to the second loop nest, and solve Equation (9) 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 $r_{12} = r_{22} = 1$. A suitable loop transformation matrix satisfying this is

$$R = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \Rightarrow S = R^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}.$$

Using this loop transformation matrix and the optimal memory layouts, the transformed program is shown in Figure 5(c). Notice that both the loop nests exhibit good locality provided that array U is diagonally stored in memory and array V is row-major. It should be emphasized that there is an additional transformation step that modifies this program for a language with fixed canonical (default) memory layout (see Section 3.4); but since that step is almost mechanical, we omit it here. The technique of how to fill out a partially completed transformation matrix (by taking into account data dependences) is similar to those used by Li [39] and by Bik and Wijshoff [8] among others.

We stress that the second loop nest in Figure 5(b) cannot be optimized using (linear) pure loop (e.g., [39]) or pure (linear) data (e.g., [44]) transformations alone; because, there is spatial reuse in two orthogonal directions. This simple example shows that an integrated approach might be useful for some programs.

The reuse analysis theory introduced by Wolf and Lam [56] shows us how to determine whether a given reference can be optimized for temporal locality in the innermost loop. Our current approach also takes this analysis into account; and, for the references with potential temporal reuse, the system of equations are modified accordingly.

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 *static* array references (not the dynamic count of the number of accesses to array elements) in the program 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 the same program can be used as a guide to decide which version is (statically) better 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 the 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. As an example, the locality coefficient of the program shown in Figure 5(b) (assuming column-major memory layouts) is 1 whereas that of the optimized code in Figure 5(c) is 4 under optimal layouts.

If desired, our model can also accommodate sophisticated techniques that estimate the number of misses in a given program. For example, instead of locality coefficients, we can use cache miss estimations obtained using the techniques proposed by Ferrante et al. [19] or Sarkar et al. [49].

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 a case, we end up with a system of equations of the following type, that we call a *target system*:

$$S\bar{\xi} \quad = \quad \bar{0}. \tag{11}$$

The systems given in Equations (8) and (9) are two example target systems. Here S is a matrix that contains only the *last* column entries of the inverse of the loop transformation matrix, and $\bar{\xi}$ is a vector obtained from concatenating the hyperplane vectors (representing memory layouts).[1] Our approach first brings this system into following form using elementary row interchange operations [36]:

$$\begin{pmatrix} S_I & Z_I \\ Z_{II} & S_{II} \end{pmatrix} \begin{pmatrix} \bar{\xi}_k \\ \bar{\xi}_u \end{pmatrix} \quad = \quad \bar{0}, \tag{12}$$

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 (that correspond to memory layouts) that have been

[1] In three- or higher-dimensional cases the rows of the layout matrices are put as sub-columns one after another.

```
do i = 1, N
 do j = 1, N
  do k = 1, N
   W(i,j)=U(i,k)
  end do
 end do
 do j = 1, N
  do k = 1, N
   V(k,j)=1.0
  end do
 end do
end do

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

(a)

```
do u = 1, N
 do v = 1, N
  do w = 1, N
   W(u,v)=U(u,w)
   V(w,v)=1.0
  end do
 end do
end do

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

(b)

Figure 6: **(a)** Original program. **(b)** Optimized program.

determined so far (presumably during handling the loop nests of higher costs). The other vector, $\bar{\xi}_u$, consists of entries of the hyperplane vectors that are to be determined.

After this point, our solution procedure consists of 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 derived.

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 discuss these three steps in greater 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 non-trivial 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 weights of the references obtained using 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 that appears in S_{II} may not appear in S_I. In that case, we choose a value for this element arbitrarily avoiding to pick up a zero value if all the other entries are zero (otherwise, this makes the last column of the inverse of the loop transformation matrix zero which, of course, is unacceptable).

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.

Consider the program fragment in Figure 6(a). At the highest level, this program fragment consists of two loop nests, the first of which (assumed more costly) is imperfectly nested. The compiler first applies loop fusion [59] to

```
do i = 1, N1
 do j = 1, N1
  do k = 1, N1
   U(i+j+k,k)=(V(j,k)*W(i,j))/2.0
  end do
 end do
end do

do i = 1, N2
 do j = 1, N2
  do k = 1, N2
   U(i,j)=(V(k,j+k)+V(k,k))/2.0
   W(i,j)=0.0
  end do
 end do
end do

do i = 1, N3
 do j = 1, N3
  do k = 1, N3
   X(i+j,j+k)=(W(i,j)+W(k,i+j)+U(j,k))/3.0
  end do
 end do
end do
```

(a)

```
do u = 1, N1
 do v = 1, N1
  do w = 1, N1
   U(u+v+w,w)=(V(v,w)*W(u,v))/2.0
  end do
 end do
end do

do u = -N2+1, N2-1
 do v = 1, N2
  do w = max(1-u,1), min(N2-u,N2)
   U(u+w,w)=(V(v,v+w)+V(v,v))/2.0
   W(u+w,w)=0.0
  end do
 end do
end do

do u = -N3+1, N3-1
 do v = max(-N3+1,u-N3+1), min(N3-1,u+N3-1)
  do w = max(-u+1,-v+1,1), min(N3,N3-u,N3-v)
   X(u+2w,u+v+2w)=(W(w,v+w)+W(v+w,u+2w)
      +U(u+w,v+w))/3.0
  end do
 end do
end do
```

(b)

Figure 7: **(a)** Original program. **(b)** Optimized program.

convert this to a perfect nest. Let, as before, Q and R be the inverses of the loop transformation matrices for the first and second loop nest, respectively. Assume that $\bar{g} = (\alpha_g, \beta_g)$, $\bar{h} = (\alpha_h, \beta_h)$, $\bar{k} = (\alpha_k, \beta_k)$, and $\bar{l} = (\alpha_l, \beta_l)$, represent the memory layouts for arrays U, V, W and X, respectively. Thus, we have the following homogeneous system:

$$\begin{pmatrix} q_{13} & q_{33} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & q_{33} & q_{23} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & q_{13} & q_{23} & 0 & 0 \\ 0 & 0 & r_{13}+r_{23} & r_{13}+r_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & r_{23}+r_{33} & r_{33} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & r_{13}+r_{33} & r_{33} \end{pmatrix} \begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \\ \alpha_k \\ \beta_k \\ \alpha_l \\ \beta_l \end{pmatrix} = \bar{0}.$$

We first divide this system into two parts. For the first nest, we obtain the following target system:

$$\begin{pmatrix} q_{13} & q_{33} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & q_{33} & q_{23} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & q_{13} & q_{23} & 0 & 0 \end{pmatrix} \begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \\ \alpha_k \\ \beta_k \\ \alpha_l \\ \beta_l \end{pmatrix} = \bar{0}.$$

Setting Q to the identity matrix, we derive $\bar{\xi}_k = (1,0,0,1)$ and $\bar{\xi}_u = (\alpha_k, \beta_k, \alpha_l, \beta_l)$. For the second loop nest, we have

$$\left(\begin{array}{cccc|cccc} 0 & 0 & r_{13}+r_{23} & r_{13}+r_{33} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & r_{23}+r_{33} & r_{33} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & r_{13}+r_{33} & r_{33} \end{array}\right)\left(\begin{array}{c} 1 \\ 0 \\ 0 \\ 1 \\ \hline \alpha_k \\ \beta_k \\ \alpha_l \\ \beta_l \end{array}\right) = \bar{0},$$

which gives us $r_{13} + r_{33} = 0$, which in turn results in $r_{13} = r_{33} = 0$ and $r_{23} = 1$ as an example solution. This result implies that $(\alpha_k, \beta_k) = (0, 1)$. With the hyperplane vectors found, we decide that array U should be row-major and arrays V and W should be column-major. The memory layout for array X, however, remains unspecified, meaning that it can be set to any layout. The reason for this flexibility is obvious from the transformed program given in Figure 6(b). Notice that the innermost loop index w in the second loop nest does not appear in the subscript functions of array X. This means that array X has temporal locality in this loop nest, and spatial locality is of secondary importance. Our method can be extended to detect that it might be preferable to choose diagonal layout for this array in order to exploit the spatial locality in the second innermost loop (the v-loop). Such an approach might be useful, especially if the trip count (the number of iterations) of the innermost loop is a very small value. We will later discuss this issue in more detail.

We finally give a slightly more complicated example to illustrate some of the problems that may occur. Consider the program fragment in Figure 7(a) that contains three loop nests and four arrays exhibiting several access patterns. Assume that $\texttt{N1} \gg \texttt{N2} \gg$ N3. Assume further that Q, R and P correspond to the inverses of the loop transformation matrices for the first (the most costly), second and third (the least costly) nest, respectively; and $\bar{g} = (\alpha_g, \beta_g)$, $\bar{h} = (\alpha_h, \beta_h)$, $\bar{k} = (\alpha_k, \beta_k)$, and $\bar{l} = (\alpha_l, \beta_l)$, represent the memory layouts for arrays U, V, W and X in that order. The homogeneous system for this program is

$$\begin{pmatrix} q_{13}+q_{23}+q_{33} & q_{33} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & q_{23} & q_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & q_{13} & q_{23} & 0 & 0 \\ r_{13} & r_{23} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & r_{33} & r_{23}+r_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & r_{33} & r_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & r_{13} & r_{23} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & p_{13}+p_{23} & p_{23}+p_{33} \\ 0 & 0 & 0 & 0 & p_{13} & p_{33} & 0 & 0 \\ 0 & 0 & 0 & 0 & p_{33} & p_{13}+p_{23} & 0 & 0 \\ p_{23} & p_{33} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} \alpha_g \\ \beta_g \\ \alpha_h \\ \beta_h \\ \alpha_k \\ \beta_k \\ \alpha_l \\ \beta_l \end{pmatrix} = \bar{0}.$$

As before, we first divide this system for each nest obtaining three target systems. For the first nest, assuming that no loop transformation will be performed, we obtain $\alpha_g = 1$, $\beta_g = -1$, $\alpha_h = 1$, and $\beta_h = 0$. Notice that, although the array W is referenced, no layout has been determined for it, because it has temporal locality in the innermost loop under the given loop order.

For the second loop nest, on the other hand, we have

$$\left(\begin{array}{cccc|cccc} r_{13} & r_{23} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & r_{33} & r_{23}+r_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & r_{33} & r_{33} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & r_{13} & r_{23} & 0 & 0 \end{array}\right) \left(\begin{array}{c} 1 \\ -1 \\ 1 \\ 0 \\ \hline \alpha_k \\ \beta_k \\ \alpha_l \\ \beta_l \end{array}\right) = \bar{0}.$$

This system is already in the desired form. Solving it gives $r_{13} = r_{23} = 1$, and $r_{33} = 0$; and $\alpha_k = 1$, and $\beta_k = -1$. Lastly, for the third loop nest,

$$\left(\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 0 & 0 & p_{13}+p_{23} & p_{23}+p_{33} \\ 0 & 0 & 0 & 0 & p_{13} & p_{33} & 0 & 0 \\ 0 & 0 & 0 & 0 & p_{33} & p_{13}+p_{23} & 0 & 0 \\ p_{23} & p_{33} & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \left(\begin{array}{c} 1 \\ -1 \\ 1 \\ 0 \\ 1 \\ -1 \\ \alpha_l \\ \beta_l \end{array}\right) = \bar{0}.$$

This system is *not* in the desired form; therefore, we apply row operations [36] and obtain

$$\left(\begin{array}{cccccc|cc} p_{23} & p_{33} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & p_{13} & p_{33} & 0 & 0 \\ 0 & 0 & 0 & 0 & p_{33} & p_{13}+p_{23} & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & p_{13}+p_{23} & p_{23}+p_{33} \end{array}\right) \left(\begin{array}{c} 1 \\ -1 \\ 1 \\ 0 \\ 1 \\ -1 \\ \hline \alpha_l \\ \beta_l \end{array}\right) = \bar{0}.$$

In solving this system, we end up with three equations

$$\{p_{23} - p_{33} = 0; \qquad p_{13} - p_{33} = 0; \qquad \text{and } p_{33} - p_{13} - p_{23} = 0\}.$$

Unfortunately, the last equation has a *conflict* with the other two; therefore we ignore it. This gives us the solution $p_{13} = p_{23} = p_{33} = 1$, which in turn results in $\alpha_l = 1$ and $\beta_l = -1$. This means that arrays U, W and X should have diagonal memory layout whereas the array V should should be row-major. The inverses of the transformation matrices used for the second and the third nests are

$$R = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \text{ and } P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \text{ respectively.}$$

The transformed program is given in Figure 7(b). Notice that with the optimized memory layouts, the spatial locality is very good except for the reference $W(v + w, u + 2w)$ in the third loop nest. This is due to the equation that we ignored while optimizing this nest.

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 finding integer solutions to non-linear systems of equations (see Section 4). If the search spaces for data and/or loop transformations are restricted, then an exhaustive search (although still costly) might be attempted [15]; but there is no experimental evidence that such an approach is fast in practice. We, instead, insist on the 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 [27].

However, for some programs it might be the case that the best optimized program[2] 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 (see Section 4.1) 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 be able to proceed to optimize each of the remaining nests using our integrated approach that employs both loop and data transformations as explained, taking the layout constraints into account. Figure 8 shows our approach. It is assumed that the nests are ordered (from top to bottom) according to non-increasing values of their weights (costs). Of course, it is possible to generalize this approach and consider different (pre-determined, three or more) layout combinations for the first (most costly) nest. However, our experience and experiments show that in practice it seems sufficient to consider only two alternatives for this nest.

To see an example for which such an approach might be useful, consider the program shown in Figure 9(a). After applying a global locality approach, either of the programs shown in Figures 9(b) and (c) can be obtained depending on how the most costly (assuming first) nest is optimized. If only data transformations are used for the most costly nest, the compiler decides row-major layout for array W; then in the second nest, it interchanges two loops, obtaining the code shown in Figure 9(b). The locality coefficient of this code is 8, all of which originates from spatial locality. On the other hand, if we optimize the first nest using loop transformations alone (assuming fixed column-major layouts), we apply a loop interchange. The second loop nest is left as it is (see Figure 9(c)). As before, the locality coefficient is 8, but this time the contribution from temporal locality is 4. Everything else being equal, we prefer the program in Figure 9(c) over the one in Figure 9(b) as it exploits more temporal reuse in the innermost loop.

It should be noted that given the fact that the global locality optimization problem is NP-complete [31], 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.

[2]Note that given the fact that the problem is NP-complete and we use a fast heuristic, our approach is not guaranteed to produce the best transformations. We believe, however, that our approach in general results in near-optimal solutions while retaining compile time efficiency.

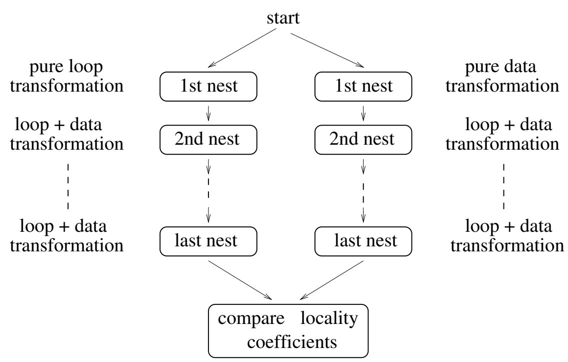

Figure 8: Extended approach that considers two alternatives for the most costly nest.

4.4 Interaction between locality and parallelism optimizations

It is also important to study the interactions between locality optimization techniques presented in this paper and the parallelism decisions that will be made by the compiler. This issue has two important sub-problems. First, the interplay between cache locality and *processor locality* (i.e., ensuring that an access by a processor can be satisfied from local memory) needs to be examined. It should be noted that our approach is oriented toward obtaining spatial and temporal locality in the innermost loop. The impact of this is that when a block is brought into cache, it will be reused as much as possible before being discarded into memory. This, in turn, results in savings in cache miss rates. However, it should be noted that this behavior will also be observed in larger data granularity on NUMA (non-uniform memory access) machines. For example, (assuming that our optimizations have been applied) when a processor brings a data page into its local memory, due to stride one accesses, it will reuse it as much as possible; that is, our approach will lead to a better processor locality in addition to better cache locality. The net result of this is significant reductions in TLB misses and page faults.

From the preceding discussion, it is possible to conclude that programs that exhibit good cache locality do not need (aggressive) explicit data placement techniques on shared memory NUMA architectures. This argument is especially true for architectures that support some kind of page migration policy (e.g., the SGI Origin 2000). Because, in these architectures, when a processor uses a data page frequently, the page is either replicated onto that processor's memory or migrated into it. In either case, most of the remaining accesses will be local. Our experimental results on the Origin 2000 also confirm this argument.[3]

Now an interesting question is that whether the programs that exhibit good processor locality need cache optimization techniques. After all, there are a number of powerful automatic data distribution techniques published in the literature (see for example [40, 5, 14, 21, 33, 47, 52] and the references therein), and for example, the SGI Origin gives the programmer fine-grain control over data distribution, that can be optimized using any of the techniques mentioned. Our answer to this question, however, is *no*; that is, just ensuring good processor locality does not imply good cache locality. It is easy to see that a processor may access only local data, but if the access stride is large then the cache locality will be poor. Our conclusion is that independent of the extent of processor locality, the compiler should try to optimize for better cache performance. Put another way, having good processor locality (as a result of careful data distribution) does not obviate the need for cache locality enhancing optimizations.

[3]It is interesting to observe that the commercial compiler writers for the NUMA architectures also share our position. For example, the SGI MIPSpro Fortran 77 Programmer's Guide [51] states that "Cache behavior continues to be the largest single factor affecting performance, and programs with good cache behavior usually have little need for explicit data placement."

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

do j = 2, N
 do i = 2, N
  W(i,j)=W(i,j-1)+W(i-1,j)
    +U(j)+V(j)
 end do
end do
```
(a)

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

do u = 2, N
 do v = 2, N
  W(u,v)=W(u,v-1)+W(u-1,v)
    +U(v)+V(v)
 end do
end do
```
(b)

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

do u = 2, N
 do v = 2, N
  W(v,u)=W(v,u-1)+W(v-1,u)
    +U(u)+V(u)
 end do
end do
```
(c)

Figure 9: **(a)** Original program **(b-c)** Optimized programs.

The second important problem is understanding the interaction between locality optimizations and loop-level parallelism decisions. It is well known that obtaining large granularity parallelism (e.g., parallelizing only the outermost loops) is beneficial for shared memory parallel architectures as it reduces synchronization and coherence activity [5, 53]. Our technique described so far also helps compiler to obtain large granularity parallelism as follows. We note that our approach attempts to optimize locality in the innermost loops using a proper mix of loop and data transformations. This approach generates—as a byproduct—outer loops that carry no reuse (hence no data dependence) and that are perfect candidates for parallelization. This is very desirable as otherwise parallelizing a loop that carries reuse is one of the main causes for inter-processor data sharing [39]. Intuitively, the more aggressive the compiler is in bringing the loops carrying reuse into innermost positions, the less the degree of true and false sharing. In fact, in our experiments we adopt this strategy; that is, given a loop nest we first apply locality optimization techniques, and after the new loop nest is obtained, we try to parallelize the outermost loops only. This strategy succeeds in general, due to the fact that after the locality transformations the outermost loops do not carry data dependences; therefore, parallelizing them does not cause excessive inter-processor communication.

4.5 Extension to higher dimensional arrays

We now briefly discuss how the optimization process handles the arrays of three or higher dimensionality. We start with the following result.

Result 1 *Let $\mathcal{C}$ be the layout matrix for a reference (to an m dimensional array) whose access matrix is $\mathcal{L}$ and let $\bar{q_n}$ be the last column of the inverse of the $n \times n$ loop transformation matrix. In order to have spatial locality in the innermost loop, the condition*

$$\mathcal{C}\mathcal{L}\bar{q_n} = \bar{0}$$

should be satisfied. Here, $\mathcal{C}$ is $(m-1) \times m$, $\mathcal{L}$ is $m \times n$, and $\bar{q_n}$ is n-dimensional; consequently, the right hand side is an $(m-1)$ dimensional zero-vector.

Proof 1 *Let $\bar{I}$ and $\bar{I}_{next}$ be two 'consecutive' iteration vectors 'after' the transformation; that is, $\bar{I}_{next} - \bar{I} = (0, \cdots, 0, 1)^T$. Also let $T = Q^{-1} = [\bar{q_1}, \bar{q_2}, \cdots, \bar{q_n}]^{-1}$. Then the array elements accessed by $\bar{I}$ and $\bar{I}_{next}$ are $\mathcal{L}Q\bar{I}$*

Table 1: Programs used in our experiments. The SIZE column gives (in terms of double precision elements) the maximum dimension size of any array in the respective program whereas the ITER column shows how many times the outermost timing loop has been iterated for each code.

PROGRAM	SOURCE	SIZE	ITER	NUMBER OF ARRAYS
mxm	Spec92/Nasa7	980	20	three 2-D
adi	Livermore	2,048	2	three 1-D, three 3-D
vpenta	Spec92/Nasa7	920	20	seven 2-D, two 3-D
btrix	Spec92/Nasa7	150	25	twenty-five 1-D, four 4-D
syr2k	BLAS	2,048	1	three 2-D
htribk	Eispack	1,200	1	five 2-D
gfunp	Hompack	2,048	20	one 1-D, five 2-D
trans	NWChem	4,000	10	two 2-D

Table 2: Different versions of the codes used in our experiments. For the LOP version we used the technique given by Li or let the native compiler to derive an order whereas for the DAT version we used an approach that uses only data transformations to optimize spatial locality (without directly exploiting temporal locality). The INT version is the version that is obtained by applying our integrated technique explained in this paper.

VERSION	BRIEF DESCRIPTION
CLM	original code: fixed column-major memory layout for all arrays
ROW	original code: fixed row-major memory layouts for all arrays
LOP	loop-optimized version: no memory layout transformation
DAT	layout-optimized version: no loop transformation
INT	our approach : integrated loop & data layout transformations

and $\mathcal{L}Q\bar{I}_{next}$, *respectively. In order to have spatial locality in the innermost loop, the condition* $\mathcal{CL}Q\bar{I} = \mathcal{CL}Q\bar{I}_{next}$ *should hold. Solving this last equation, we get* $\mathcal{CL}Q(\bar{I}_{next} - \bar{I}) = \bar{0} \Rightarrow \mathcal{CL}Q(0, \cdots, 0, 1)^{-1} = \bar{0} \Rightarrow \mathcal{CL}\bar{q_n} = \bar{0}$. □

Notice that, depending on the loop trip counts, it might be desirable to exploit the locality in the second innermost loop as well. This corresponds to determining the second rightmost column of Q in addition to its rightmost column. It is easy to show that in this case the condition $\mathcal{CL}[\bar{q}_{n-1}, \bar{q_n}] = [0]$ should be satisfied. Here the right hand side zero matrix is of size $(m-1) \times 2$. This result can easily be generalized to the outer loops as well.

We should stress that there is a subtle problem in determining both $\bar{q}_{n-1}$ and $\bar{q_n}$ together. It is known from linear algebra [36] that if we determine only $\bar{q_n}$ and if the greatest common divisor (or *gcd*) of its entries is one (which is usually the case), then it is always possible to complete this $\bar{q_n}$ to a unimodular matrix Q (and, of course, T will also be unimodular). If we can achive the same objective (which might be obtaining a target amount of reuse in the innermost loop) with a unimodular matrix instead of a non-unimodular matrix, that is good; because, in general, code generation after unimodular transformations is easier and the resultant code is more efficient than the non-unimodular case [39]. However, if we determine both $\bar{q}_{n-1}$ and $\bar{q_n}$, the resultant Q matrix (after completion) may or may not be unimodular. This tradeoff between aggressive optimization in multiple loop levels, and the ease and efficiency of code generation is an interesting one; but, given current optimizing compiler technology it seems difficult to resolve it fully at compile-time. Notice that another way of exploiting locality in outer loops is to apply tiling. We will briefly discuss the interaction between tiling and our optimization technique in the next section.

Table 3: Performance summary of the CLM (original) version on *eight* processors. The CYCLES column gives the total processor cycles spent in executing the code. The LOADS and STORES columns present the number of memory load and store operations. PC, SC, and TLB refer to primary cache, secondary cache, and TLB misses, respectively. The TIME column gives the total execution times in *seconds*. All other entries are in *millions*.

PROGRAM	CYCLES	LOADS	STORES	PC	SC	TLB	TIME
mxm	39,397	4,953	712	192	39.6	0.34	44.39
adi	16,041	572	184	144	73.6	42.9	63.87
vpenta	18,266	6,240	244	244	14.2	69.0	24.31
btrix	13,672	4,818	41.4	12.8	2.45	8.86	22.11
syr2k	55,131	8,246	310	865	244	211	69.13
htribk	2,257	639	140	175	1.94	20.9	16.76
gfunp	7,330	800	162	267	26.3	0.15	19.28
trans	16,867	2,188	663	207	17.4	171	17.20

5 Experimental results

In this section we present performance results to demonstrate the impact of our global locality optimization approach. Our experimental platform is an eight node SGI Origin 2000 at the Center for Parallel and Distributed Computing at Northwestern University. This machine uses 195MHz R10000 processors, each with a 32KB L1 data cache and a 4MB L2 unified cache. The processors 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 four outstanding misses from the combined two levels of cache are supported. The R10000 processor dynamically schedules instructions whose operands are available in order to hide the latency of cache misses. For the L1 cache hits, the latency is 2 cycles; and for L1 misses that hit in the L2 cache, the latency is 8 to 10 cycles. The non-local accesses take at most 20 cycles.

We experiment with eight programs from benchmarks and libraries whose important characteristics are listed in Table 1. The SIZE size column gives the maximum dimension size of any array used in the program. In order to fully investigate the impact of the locality optimizations the dimension sizes are increased from their default values. However, some hard-coded dimension sizes (e.g., with a fixed value of 4 or 5) are not modified as modifying them would require a complete understanding of what the applications perform. The ITER column, on the other hand, shows how many times the outermost timing loop is iterated for each code. Note that in order to see the impact of the additional power provided by integration of loop and data transformations on well-optimized codes we also include two programs from two libraries, Eispack and Hompack.

For each program in our experimental suite, we perform experiments with five different *versions* briefly summarized in Table 2. The first two versions, CLM and ROW are the original programs; only the layouts of the arrays are different. In the CLM version, all the arrays have column-major memory layout (as in Fortran) while in the ROW version, all the arrays have row-major memory layout (as in C). For the LOP version we use the better of the results from the technique given by Li [39] and the *native* optimizing compiler. For the DAT version, we apply a technique proposed in [27] that is based on *pure* data transformations. In effect, for the programs in our experimental suite, other pure data transformation techniques proposed in the literature such as O'Boyle and Knijnenburg [44] and Leung and Zahorjan [38] result in the same output codes as ours. Finally, the INT version is the one that is obtained by applying our integrated technique explained in this paper.

We first hand-coded the C versions of the programs in our experimental suite. Then the programs were transformed automatically for each specific version using a compiler front-end built on top of the Omega library [32]. Then, the output C codes are compiled using the native C compiler using the -O2 option. We have noted that applying our integrated approach increased the overall compilation time by at most 16% (not including the time spent in profiling as in these codes a simple static analysis was sufficient to order the the loop nests according to their costs). For each nest in the programs, we parallelized only the outermost loop that does not carry any data dependence. Note that this is a locality-based parallelization strategy; that is, we first optimize for locality and then parallelize the outermost loop in the resultant nest. We modified this strategy only for the LOP version in the cases where the native compiler derives a better code balancing locality and parallelism. We report results showing the numbers of cycles, loads, stores as well as the primary cache, secondary cache, and the TLB misses. Unless stated otherwise, all the reported numbers are *cumulative*; that is, summed over all the processors involved. Table 3 presents the performance summary for the original version (CLM) of each code on *eight* processors. The TIME column gives the total execution time in seconds. The figures given in this table (except those under the TIME column) are in millions and form a base for comparison of the figures to be presented shortly.

We first show in Figure 10 the static improvements achieved by our approach. This figure shows for each version of each code the ratio of the locality coefficient (see Section 4.1) to the total number of references. This ratio is between 1 and 0, depending on whether the code exhibits good locality in the *innermost* loop or not. We note that in all cases (except btrix) the INT version optimizes all references in the programs for either spatial or temporal locality in the innermost loop. In btrix, conflicting access patterns to the same array prevent our technique from exploiting locality fully for all the nests. We note that the DAT version is also quite successful. These static results, however, are not very conclusive as it may be important to distinguish between spatial and temporal locality and to distinguish between sizes of the arrays with locality. Nevertheless, the results show that our approach optimizes the references successfully for better locality.

Next we present results on *eight* processors of the Origin. Except for megaflop rates (which are obtained using timing routines in the program) all the other numbers to be presented are obtained using *hardware performance counters* on the machine. Figure 11 shows the overall processor cycles for each program. The results shown in this figure and the results to be presented in the following are all *normalized* according to the *worst–performing version* (whatever version it might be) for each program. It is easy to see from Figure 11 that the INT version is quite successful, and in all cases achieves the best results. Figures 12 and 13, on the other hand, give the normalized number of load and store operations. The results reveal that as compared to the original (CLM) version in all cases our approach reduces the number of loads, and only in one program it increases the original number of stores. These results are good, because, locality optimizations in some cases can increase the number of load/store operations as most of them are oriented for optimizing spatial locality rather than temporal locality. For instance, the best performing loop order known for the classical i, j, k matrix-multiply nest increases the original number of stores substantially (see [39]). As explained earlier, our approach is able to take temporal locality into account as well.

Figures 14, 15 and 16 present results about the memory behavior of the different versions. In short, the improvement with our locality enhancing technique originates from *all* levels of memory hierarchy; that is, our approach improves cache as well as processor locality. From Figure 14 we see that in programs like vpenta, syr2k, and btrix, we achieve impressive reductions in primary cache misses. As compared to the DAT version, our approach increases the primary cache misses only in adi (in this code the benefit comes from the reduction in the secondary cache misses). We should also note that although the DAT version is successful, the INT version outperforms it with a large margin in codes like vpenta, btrix, and gfunp. As far as the secondary cache misses are concerned, our approach achieves the best result in four out of eight programs.

The impact of our approach on TLB misses is very significant. As can be seen from Figure 16, for the first four codes in our experimental suite, our approach eliminates almost all TLB misses, confirming our argument on processor locality in Section 4.4. Notice, however, that this picture of the TLB misses can be misleading; as the TLB misses are

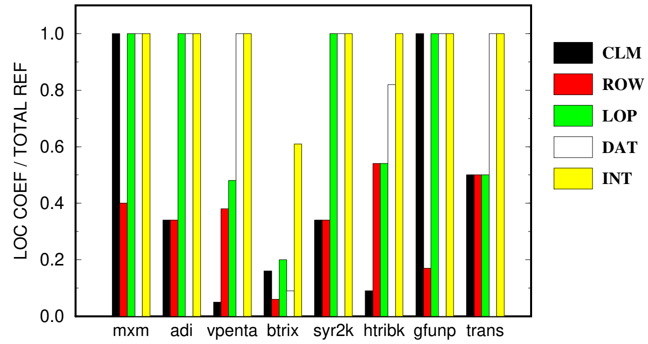

Figure 10: Summary of the static performance of different versions. This figure shows (for each version of each code) the ratio of the locality coefficient to the total number of references. This ratio is between 1 and 0, depending on whether the code exhibits good locality in the innermost loop or not. We note that in all cases (except `btrix`) the `INT` version optimizes all references in the programs for locality in the innermost loop.

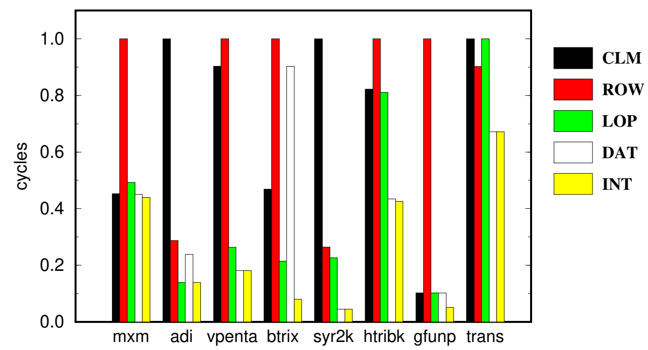

Figure 11: Processor cycles. The results shown are normalized according to the worst–performing version for each program. Note that the `INT` version is quite successful, and in all cases achieves the best results.

not in general as numerous as the cache misses (see Table 3) and on the Origin that we use the non-local misses can take at most twice time as the local misses. Moreover, R10000 is a very complex processor and a lot of misses can get overlapped with the ongoing computation activity within the processor. Therefore, it is important to have a look at the overall performance depicted in Figure 17.

In Figure 17 we present the absolute megaflop rates of our programs for the different versions on two processors. As compared with the original code (`CLM`), the performance of the `LOP` version degrades in two cases, and results in the same performance as the original in two others. The `DAT` version is more successful and degrades the performance in only one code, and results in the original code in another. It is important to see that neither `LOP` nor `DAT` dominates the other, and our approach (`INT`) achieves the best results for all programs. In three of the codes (`vpenta`, `syr2k`, and `trans`), the `DAT` version obtains the same performance as ours; and only in the `adi` code, `LOP` and `INT` generates the same output code. These results are promising and we believe that our approach is more successful than the current *linear* transformation techniques for optimizing locality.

We now study scalability using two example programs: `btrix` and `trans`. The reason that we use these two programs is that they demonstrate two distinct representative behaviors that we have observed during the experiments.

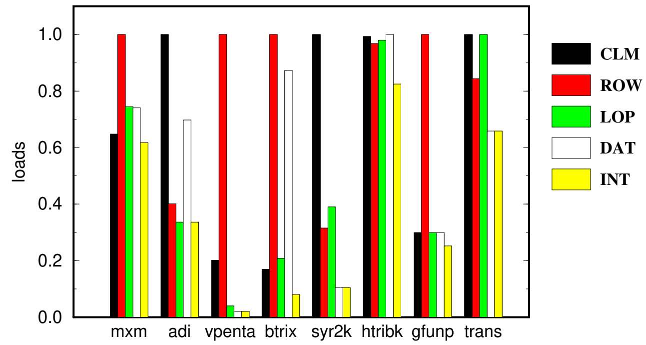

Figure 12: Memory loads. The results shown are normalized according to the worst–performing version for each program. As compared to the original (`CLM`) version, our approach (`INT`) reduces the number of loads in all programs.

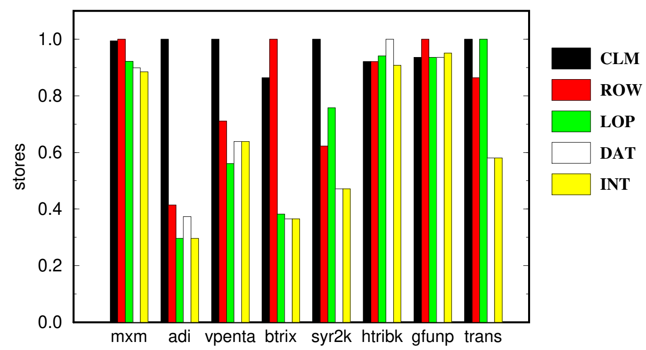

Figure 13: Memory stores. The results shown are normalized according to the worst–performing version for each program. As compared to the original (`CLM`) version, our approach (`INT`) increases the number of stores in only one program.

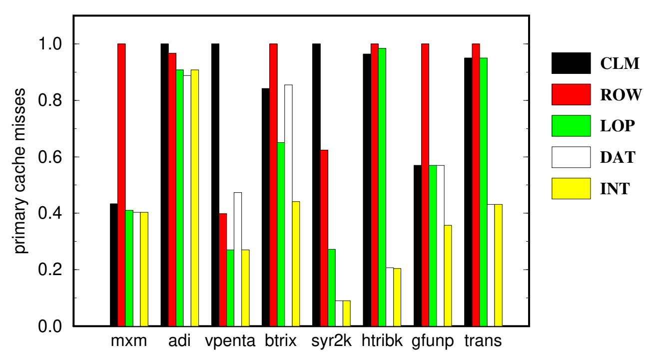

Figure 14: Primary cache misses. The results shown are normalized according to the worst–performing version for each program. As compared to the `DAT` version, our approach increases the primary cache misses only in `adi`. Note that although the `DAT` version is successful, the `INT` version outperforms it with a large margin in codes like `vpenta`, `btrix`, and `gfunp`.

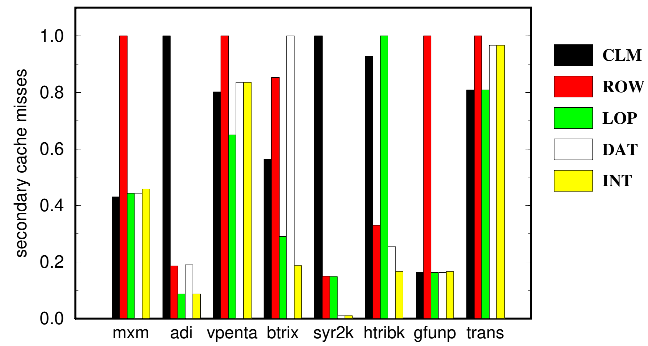

Figure 15: Secondary cache misses. The results shown are normalized according to the worst–performing version for each program. Our approach achieves the best result in four out of eight programs.

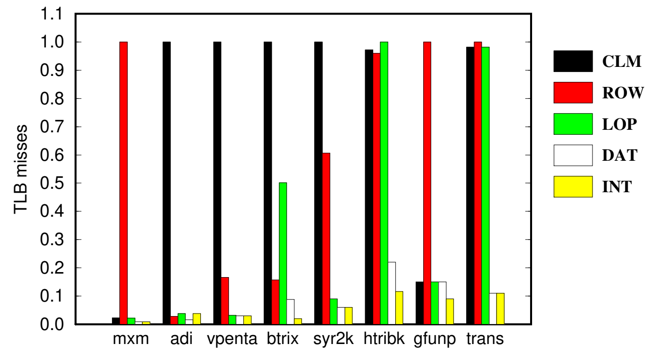

Figure 16: TLB misses. The results shown are normalized according to the worst–performing version for each program. The impact of our approach on TLB misses is very significant. For the first four codes in our experimental suite, our approach eliminates almost all TLB misses, confirming our argument on processor locality in Section 4.4.

In other words, the scalability of any one of the remaining codes is similar to that of one of these two programs. Figure 18(a) shows the performance of `btrix` on different number of processors. The y-axis shows (in millions) the cycles *per processor*. The problem size is 150 as before, but we run only a single iteration. It can be observed from the figure that the performance of the original code (`CLM`) is not stable. The performance of `ROW` and `DAT` is very good on a single node; but when the number of nodes is increased, their performance degrades drastically due to false sharing. In fact, this underlines an important limitation of pure data transformations: while they optimize locality on single node very well, since they can not take false sharing (or other parallelism-related factors) into account they may perform very poorly on multiple node case. The performance of the `LOP` version is good but starts to degrade beyond six nodes. Finally, we see that `INT` performs well and outperforms the rest of the versions for all processor sizes. We expect this behavior to be prevalent in larger number of nodes as well. Figure 18(b), on the other hand, shows the performance of the `trans` code with 2048×2048 double precision arrays. This illustrates the second kind of behavior that we have observed in our experiments: the original program scales relatively well. In that case, all versions scale well and the optimized code (`INT`) outperforms the original version with almost a fixed margin for each processor size. It should also be mentioned that previous combined (loop+data) transformation techniques use a limited set of loop as well as data transformations (e.g., [31] and [15]), and cannot optimize the programs that require diagonal memory layouts as

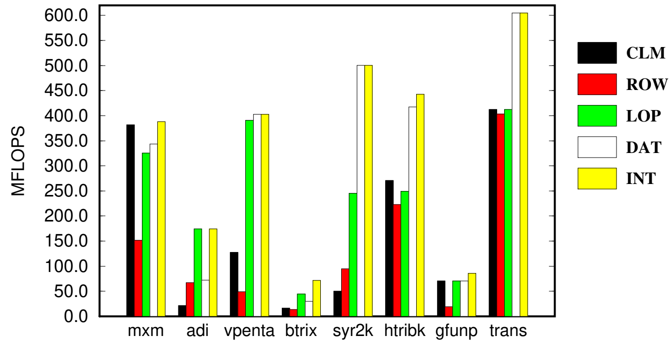

Figure 17: MFLOPS rates. Note that neither `LOP` nor `DAT` subsumes the other, and our integrated approach (`INT`) achieves the best results for all programs.

in `syr2k` and `gfunp`. For the remaining programs in our experimental suite, we can expect the approaches proposed in [15] and [31] to generate similar results. However, since how these approaches handle global locality optimization problem is not described fully, a complete comparison is not possible.

Finally, we discuss briefly the interaction between our locality optimization technique and tiling, which is a combination of strip-mining and loop permutation [58, 56, 43]. In its most general form tiling replaces an n-deep loop nest with a new loop nest of depth $2n$. The outermost n loops enumerate individual tiles whereas the innermost n loops execute the iterations of a given tile. It is well-known that tiling can improve cache locality for a given loop nest significantly by exploiting the reuse in outer loops [43]. There are, however, several problems with tiling. First, like other loop based transformation techniques, tiling is constrained by data dependences; that is, it is not always legal to tile a given loop nest. Second, it is not very easy to select a good tile size. Previous studies have shown that the performance of tiling is *very sensitive* to the tile size and a wrong tile size in fact can degrade the performance [56, 17, 37]. In that regard, we believe that our approach is a suitable step prior to tiling. This is because, as observed by Li [39], improving spatial locality before tiling improves the inter-tile locality thereby reducing the sensitivity of tiling to the tile size. Therefore, our optimization strategy helps tiling to achieve better performance by making its performance almost independent from the tile size. In this respect, we go one step further from Li, though. Because, since we use data transformations as well, our linear locality optimization approach is more aggressive than Li's. Consider now Figure 19 that shows the performances of two *tiled* programs on a single node of the Origin with different tile size selections. The codes considered are the classical i, j, k matrix-multiply routine and `trans`. For the matrix-multiply code we use $1,024 \times 1,024$ double precision arrays. Note that tile size = $1,024$ corresponds to *no tiling*. It is easy to see from Figure 19(a) that the performance of the naive tiling (i.e., without first applying locality optimizations) is very sensitive to the tile size. While the performance is good until tile size = 128, beyond that size it degrades drastically. This observation is consistent with Li's findings [39] as well as Lam et al.'s [37] suggestion of using small tile sizes when precise performance analysis is not available. When we first optimize spatial and temporal locality and then tile the loops, however, the picture totally changes. All the tile sizes between 16 and 512 behave almost similar and equally good. Of course, the tile sizes between 512 and $1,024$ are not reasonable given the fact that the arrays are $1,024 \times 1,024$. The moral of the story is that optimizing locality before tiling almost obviates the need for analysis to select a suitable tile size. We also stress that the analysis for tile size selection can be quite costly. For example, Wolf et al. [57] derive a cost formula as a function of the unknown tile sizes. And then they attempt to select a tile size that minimizes this function. Figure 19(b) shows the performance of the tiled versions of the original (`CLM`) and optimized

(INT) versions of the trans code. The arrays sizes are $4,096 \times 4,096$ double precision elements. Although for this example tiling does not seem very beneficial in general, it is important to note two points. First, for the CLM version, as in the matrix-multiply code, beyond a certain tile size (32) the performance starts to degrade (not so drastically, though) whereas the tiled INT version's performance is quite stable. Secondly, in contrast to the matrix-multiply nest, no *tiled* version of CLM achieves the same performance as the *untiled* INT (i.e., tiles size = $4,096$) code. We believe that these two example underline the importance of locality optimizations before tiling. Lastly, by increasing the number of loops, tiling increases the run-time overhead of nest execution. Our approach is useful in that respect too as we try to place all the spatial locality in the innermost loops and obviate the need for tiling the outermost loops that do not carry any type of reuse (in the examples mentioned above, however, we tiled all the loops). That is, our approach helps a compiler to apply tiling more judiciously. A complete treatment of the interplay between locality enhancing (linear) techniques and tiling is considered as possible future work.

6 Related work

Several researchers have focused on the problem of exploiting memory hierarchy by restructuring programs. A majority of these restructuring techniques is based on iteration space transformations.

McKellar and Coffman [42] performed one of the first studies on program transformations for locality. They showed that by using sub-matrix operations it is possible to obtain impressive speedups over the original matrix codes. Later Abu-Sufah et al. [1] focused on automating page locality improving techniques within a compilation framework, and discussed a transformation technique called *vertical distribution*, which is very similar to tiling.

In his dissertation, Porterfield [46] uses loop transformation techniques such as skewing and tiling. His main objective is to model fully-associative caches with a least recently used (LRU) policy. Like Gannon et al. [20], he focuses on estimating the cache miss rates for a given loop nest. These approaches, however, do not propose how to reach the best transformed version, and imply that a number of candidate solutions should be evaluated. The works of Ferrante et al. [19] and Sarkar et al. [49] can also be considered to belong to this category.

Wolf and Lam [56] describe *reuse vectors* and explain how they can be used for optimizing cache locality. Their approach involves first optimizing nest-locality using uni-modular loop transformations and then applying tiling to the loops that carry some type of reuse. Their method uses a sort of exhaustive search and in some cases can only work with the approximate reuse vectors. Li [39] also considers reuse vectors but determines an appropriate loop transformation matrix in one go rather than resorting to an exhaustive search. He shows that optimizing spatial locality before tiling *lessens* the sensitivity in performance of the tiling to the tile size. Neither Li [39] nor Wolf and Lam [56] consider memory layout transformations; and since a loop transformation to improve locality of a reference can sometimes adversely affect the locality of another reference in the nest, both approaches may end up with unsatisfactory solutions for a given loop nest.

The cost of the methods mentioned is partly eliminated by a simple heuristic used by McKinley et al. [43]. Arguing that the general non-singular loop transformations may not be necessary for many codes, they propose a method that employs a simple cost formulation and considers loop permutation, loop reversal, loop fusion, and loop distribution (fission). In addition to having the disadvantages of an approach that 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 (e.g., the syr2k code from Blas).

Considering the fact that linear loop transformations may be insufficient for some loop nests, some researchers have focused on loop tiling [58, 37] which in most cases can be accomplished via a combination of strip-mining and loop interchanging. There are however a number of problems with tiling as explained in the previous section. It has been observed by Li [39] that in most cases applying locality enhancing linear loop transformations before tiling improves its performance, in some cases significantly. Our own experiments reported in the previous section also confirm Li's findings and underline the importance of optimizing spatial locality before tiling.

Recently some researchers have considered data layout transformations that are simply restructuring of multi-dimensional arrays in memory as an alternative to the loop transformations. Jeremiassen and Eggers [25] use data transformations for eliminating the effect of false sharing, whereas Ju and Dietz [26] use them for reducing the coherence activity on shared memory multiprocessor systems.

More recent work on data transformations is on optimizing cache locality. Leung and Zahorjan [38] present a technique that 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. They also handle the problem of minimizing the extra memory requirements induced by a given data transformation. In this paper, we have not attempted to attack the problem of minimizing the extra space;[4] however, if necessary, we can use their techniques. In comparison, our approach uses both loop an data transformations and is more powerful than the one proposed in [38].

O'Boyle and Knijnenburg [44] 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. Their main concern, however, is to handle code generation after a data layout transformation.

Kandemir et al. [27] also propose a layout optimization technique. They use layout matrices to represent the memory layouts of multi-dimensional arrays. In this paper, we show that how this explicit representation of layouts helps to combine loop and data transformations in a unified framework.

Anderson et al. [4] propose a transformation technique that makes the data elements accessed by the same processor contiguous in the shared address space. Their method is mainly for shared-memory parallel architectures. They use only permutations (of array dimensions) and strip-mining for possible data transformations. Our work is more general as we consider a much larger search space for possible layout transformations. As far as the multiprocessors are concerned, both approaches can be useful for reducing false sharing.

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 all the references (some perhaps adversely) to that array in all the 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 that satisfies as many nests as possible. Leung and Zahorjan [38] and Kandemir et al. [27] handle this multiple-nest problem by enclosing all the loop nests with an imaginary outermost loop that iterates only once. Unfortunately, this technique may not be very successful when there are conflicting references to the *same* array. Since our approach uses iteration space transformations as well, we can handle temporal locality too. Moreover, we show how to eliminate the potential negative impact of data transformations by using loop transformations.

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 [15] 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 dimension permutations only (e.g., converting from column-major to row-major); therefore, they cannot optimize banded matrix applications fully for which diagonal layouts are the most suitable. The loop transformation matrices that they consider, on the other hand, are the ones that contain only ones and zeroes.

[4] Because, the extra memory space required by our approach was never more than 8% of the total size of the arrays in the program.

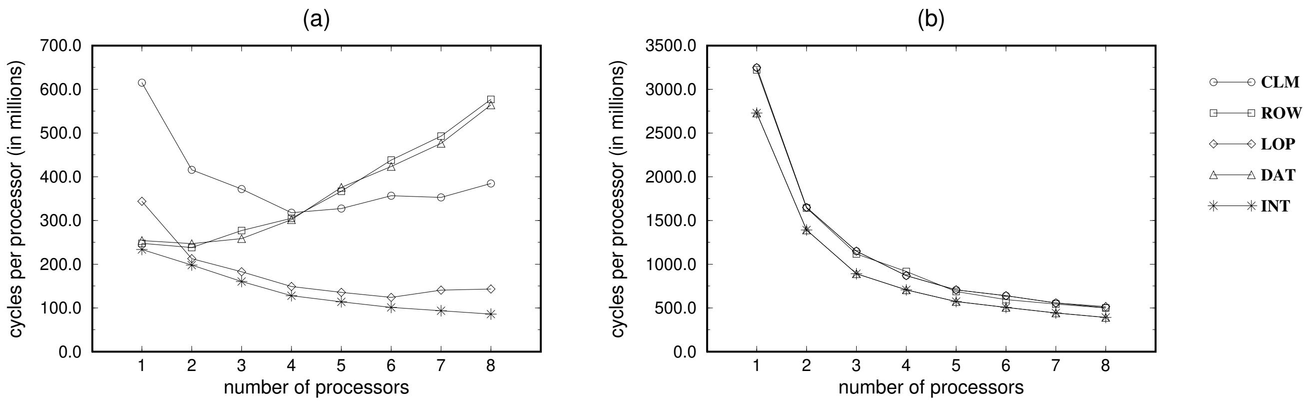

Figure 18: Scalability of **(a)** `btrix` with a single iteration of timing loops and a problem size of 150, and **(b)** `trans` with 10 iterations of the timing loop and a problem size of $2,048$.

In [30] and [31], the authors presented another approach to combine loop and data transformations. That approach does not restrict possible loop transformations; but, like Cierniak and Li [15], uses only dimension permutations as possible data transformations; therefore, the approach presented in [30] (for sequential machines) and [31] (for parallel machines) cannot optimize the nests whose arrays require diagonal memory layouts for the best cache locality.

An important drawback of the works in [15], [30] and [31] is that they use *exhaustive search* to find the solutions; and they cannot handle skewed (diagonal) memory layouts that are very useful for banded matrix applications. The approach presented in this paper has no restriction on layout transformations, and finds the solution without doing exhaustive search. 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. Lastly, rather than limiting scope to a single loop nest we focus on a sequence of loop nests and propagate memory layouts across loop nests.

Finally, we note that the global memory layout determination problem bears similarities to the automatic data distribution problem [5, 47, 21, 52] for distributed-memory machines. Unlike data distribution which is applicable only for parallel machines, memory layouts affect the performance of uniprocessor and multiprocessor machines. In addition, good global memory layouts influence decisions on data distribution.

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 considering a loop nest, our approach first applies a loop transformation to it to satisfy the layout requirements for the references to those arrays whose layouts have already been determined. It then determines suitable memory layouts for the remaining arrays referenced in the nest. For the first nest to be optimized, however, we use both loop and data transformations. Although the general problem appears to be difficult, we have shown in this paper that the whole process for a single nest can be formulated in a mathematical framework which is based on explicit memory layout representations. We have also shown that our approach is more successful than existing locality-enhancing (linear transformation) techniques whether they are pure loop-based, pure data-based, or a combination of the two.

A detailed study of the interaction between our solution and tiling is in progress. Along these lines, we plan to work

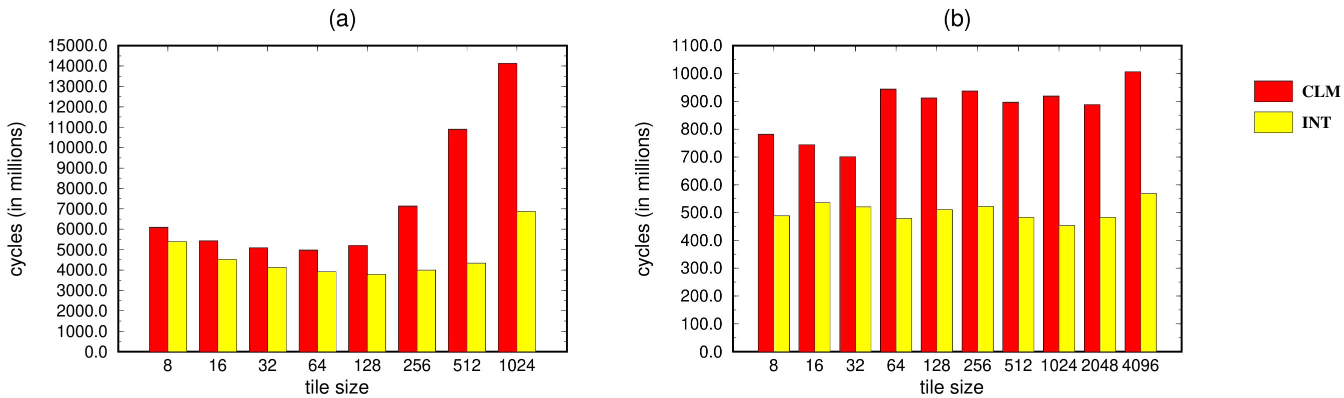

Figure 19: Impact of tile size on the performance. **(a)** matrix-multiply with $1,024 \times 1,024$ double precision arrays, and **(b)** `trans` with $4,096 \times 4,096$ double precision arrays.

on several related problems such as evaluating extensively the 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 [35]. 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. Work is also in progress on extending our techniques to optimize locality across program modules.

We believe that the experimental results reported in this paper are promising and an integrated approach that applies both loop and data transformations in concert will help optimizing compilers in exploiting the deep memory hierarchies found in current parallel architectures to the fullest extent possible.

Acknowledgments

We wish to thank the anonymous referees for their insightful comments and suggestions.

The material presented in this paper is based on research supported in part by Alok Choudhary's NSF Young Investigator Award CCR-9357840, NSF grants CCR-9509143 and CCR-9796029, the Air Force Materials Command under contract F30602-97-C-0026, and the Department of Energy under the ASCI Academic Strategic Alliance Program Level 2, under subcontract No W-7405-ENG-48 from Lawrence Livermore Labs. The work of Prith Banerjee is supported in part by the NSF under grant CCR-9526325 and in part by the DARPA under contract F30602-98-0144. The work of J. Ramanujam is supported in part by an NSF Young Investigator Award CCR-9457768.

References

[1] A. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Trans. on Computers*, C-30(5):341–356, 1981.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[3] J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors.* Ph.D. dissertation, Stanford University, March 1997. Also available as Technical Report CSL-TR-97-179, Computer Systems Laboratory, Stanford University.

[4] 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 (PPoPP'95)*, July 1995.

[5] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, pages 112–125, June 1993.

[6] D. Bacon, J-H. Chow, D. Ching, R. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proc. CASCON'94 conference*, Toronto, Canada, November 1994.

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

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

[9] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced program restructuring for high-performance computers with Polaris. *IEEE Computer*, December 1996, pages 78–82.

[10] W. Blume and R. Eigenmann. An overview of symbolic analysis techniques needed for the effective parallelization of the PERFECT benchmarks. In *Proc. 1994 International Conference on Parallel Processing (ICPP'94),* pages II.233–II.238, August, 1994.

[11] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*, ACM, New York.

[12] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst.*, 16(6):1769–1810, November 1994.

[13] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. M. Anderson. Data-distribution support on distributed-shared memory multiprocessors. In *Proc. Programming Language Design and Implementation (PLDI'97)*, Las Vegas, NV, 1997.

[14] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Optimal Evaluation of Array Expressions on Massively Parallel Machines. *ACM Transactions on Programming Languages and Systems,* 17(1):123-156, January 1995.

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

[16] M. Cierniak, and W. Li. Recovering logical data and code structures. Technical Report 591. Dept. of Computer Science, University of Rochester, July 1995.

[17] 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'95)*, June 1995.

[18] M. Dion, C. Randriamaro, and Y. Robert. Compiling affine nested loops: how to optimize the residual communications after the alignment phase. *Journal of Parallel and Distributed Computing (JPDC),* volume 38.2, pages 176–187, 1996.

[19] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proc. Languages and Compilers for Parallel Computing (LCPC'91)*, pages 328–343, 1991.

[20] 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.

[21] J. Garcia, E. Ayguade, and J. Labarta. A novel approach towards automatic data distribution. In *Proc. Supercomputing'95*, San Diego, December 1995.

[22] M. Haghighat and C. Polychronopoulos. Symbolic analysis: a basis for parallelization, optimization, and scheduling of programs. In *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'93),* Portland, OR, 1993.

[23] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Second edition, Morgan Kaufmann Publishers, San Mateo, CA, 1995.

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

[25] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, July 1995.

[26] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Proc. Languages and Compilers for Parallel Computing (LCPC'92),* U. Banerjee et al. (Eds.), Lecture Notes in Computer Science, pages 344–358, 1992.

[27] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A hyperplane based approach for optimizing spatial locality in loop nests. In *Proc. 1998 ACM International Conference on Supercomputing (ICS'98)*, July 1998.

[28] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 14–17, 1998, Paris, France.

[29] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee. An iteration space transformation algorithm based on explicit data layout representation for optimizing locality. In *Proc. Languages & Compilers for Parallel Computing (LCPC'98)*, Chapel Hill, NC, August 1998.

[30] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proc. 11th ACM International Conference on Supercomputing (ICS'97)*, pages 269–276, Vienna, Austria, July 1997.

[31] 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.

[32] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, CS Dept., University of Maryland, College Park, March 1995.

[33] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proc. Supercomputing'95*, San Diego, CA, December 1995.

[34] I. Kodukula and K. Pingali. Transformations of imperfectly nested loops. In *Proc. Supercomputing*, November 1996.

[35] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. Programming Language Design and Implementation (PLDI'97)*, June 1997.

[36] B. Kolman. *Introductory Linear Algebra with Applications*, Prentice-Hall, 1997.

[37] 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 (ASPLOS'91)*, April 1991.

[38] S.-T. Leung, and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR 95-09-01, Dept. Computer Science and Engineering, University of Washington, Sept 1995.

[39] W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Cornell University, Ithaca, NY, 1993.

[40] J. Li and M. Chen. Compiling communication efficient programs for massively parallel machines. *Journal of Parallel and Distributed Computing*, 2(3):361–376, 1991.

[41] V. Maslov. De-linearization: an efficient way to break multi-loop dependence equations. In *Proc. the SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, CA, June 1992.

[42] A. McKeller and E. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *CACM*, 12(3):153–165, 1969.

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

[44] 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, 1996.

[45] M. O'Boyle and P. Knijnenburg. Integrating loop and data transformations for global optimisation. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 14–17, 1998, Paris, France.

[46] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. Ph.D. Thesis, Rice University, Houston, May 1989.

[47] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel & Distributed Sys.,* 2(4), pages 472–482, Oct 1991.

[48] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, June 1998.

[49] V. Sarkar, G. R. Gao, and S. Han. Locality analysis for distributed shared-Memory multiprocessors. In *Proc. 9th Workshop on Languages and Compilers for Parallel Computing (LCPC'96)*, Santa Clara, California, August 1996.

[50] A. Schrijver. *Theory of Linear and Integer Programming*, John Wiley, 1986.

[51] *SGI MIPSpro Fortran* 77 *Programmer's Guide*, SGI Corporation (also available using the InSight tool on the SGI Origin 2000 machines).

[52] S. Tandri and T. Abdelrahman. Automatic partitioning of data and computations on scalable shared memory multiprocessors. In *Proc. 1997 International Conference on Parallel Processing (ICPP'97)*, Bloomingdale, IL, pages 64–73, August 1997.

[53] C.-W. Tseng, J. Anderson, S. Amarasinghe, and M. Lam. Unified compilation techniques for shared and distributed address space machines. In *Proc. ACM International Conference on Supercomputing (ICS'95)*, July 1995.

[54] 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.

[55] E. Torrie, C-W. Tseng, M. Martonosi, and M. W. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'95)*, June 1995.

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

[57] M. Wolf, D. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proc. MICRO'96*, pages 274–286, 1996.

[58] M. Wolfe. More iteration space tiling. In *Proc. Supercomputing'89*, pages 655–664, November 1989.

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