Optimizing Inter-Nest Data Locality*

M. Kandemir, I. Kadayif
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802, USA
{kandemir,kadayif}@cse.psu.edu

A. Choudhary, J. A. Zambreno
ECE Department
Northwestern University
Evanston, IL 60208, USA
{choudhar,zambro1}@ece.nwu.edu

ABSTRACT

By examining data reuse patterns of four array-intensive embedded applications, we found that these codes exhibit a significant amount of inter-nest reuse (i.e., the data reuse that occurs between different nests). While traditional compiler techniques that target array-intensive applications can exploit intra-nest data reuse, there has not been much success in the past in taking advantage of inter-nest data reuse. In this paper, we present a compiler strategy that optimizes inter-nest reuse using loop (iteration space) transformations. Our approach captures the impact of execution of a nest on cache contents using an abstraction called footprint vector. Then, it transforms a given nest such that the new (transformed) access pattern reuses the data left in cache by the previous nest in the code. In optimizing inter-nest locality, our approach also tries to achieve good intra-nest locality. Our simulation results indicate large performance improvements. In particular, inter-nest loop optimization generates competitive results with intra-nest loop and data optimizations.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory Structures; D.3.4 [**Programming Languages**]: Processors—*Compilers;Optimization*

General Terms

Design, Experimentation, Performance

Keywords

Data Reuse, Cache Locality, Inter-Nest Optimization, Array-Intensive Codes, Embedded Applications.

1. INTRODUCTION AND MOTIVATION

On-chip cache architectures are increasingly being used in embedded system designs. While caches are very effective for applications that exhibit good data locality, many large-scale embedded applications manipulate large data sets (e.g., images and video frames), incurring frequent cache misses. Given the large access latencies of off-chip memories, ensuring data locality (i.e., acceptable cache behavior) is potentially the most important performance issue.

The optimizing compiler literature is full of algorithms/techniques for improving data cache locality (e.g., see [11] and the references therein). Since many array-intensive applications spend most of their execution times in nested loops, an overwhelming majority of these techniques focus on nested loops and try to improve their locality behavior using loop (iteration space) and memory layout (data space) transformations. A common characteristic of the iteration space based techniques is that they focus on a single nest at a time. In other words, each nest is optimized (transformed) in isolation and this optimization tries to generate the best code (from the data locality perspective) considering how the arrays are accessed by the nest body. We can refer to this kind of optimization as the *intra-nest optimization.* An intra-nest optimization assumes an empty cache (before entering the nest) and restructures loops based on the data access pattern of the nest alone.

In many array-intensive embedded applications, however, loop nests that are executed one after another access the same set of arrays. That is, there might be a significant amount of data reuse between two neighboring nests. While intra-nest optimizations can be effective in optimizing each nest in isolation, they fail to capture this *inter-nest data reuse.* To take advantage of inter-nest reuse, we need to keep track of how different arrays are shared among neighboring nests. More specifically, in optimizing the access pattern of a given nest, we can consider which array elements are currently in the cache (following the execution of the previous nest), and restructure the current nest to reuse these cache-resident data elements. In this work, we call this type of code (loop) restructuring the *inter-nest optimization.*

To demonstrate how inter-nest reuse can occur and how an optimizing compiler can take advantage of it, we consider the program fragment shown below:

$$\begin{array}{l}\text{for } (i_1 = 0; i_2 < n; i_1 + +)\\ \quad \{ \ldots U[i_1] \ldots \}\\ \text{for } (i_3 = 0; i_3 < n; i_3 + +)\\ \quad \{ \ldots U[i_3] \ldots \}\end{array}$$

In this fragment, there are two separate nests (each with one loop) that access the same array. It should be noted that each loop traverses the array from the first element to the last (assuming that the array has n elements). If the data cache used cannot hold the entire array U, every element of the array will be brought from off-chip memory to cache twice: once in the first loop and one more in the second loop. This is because if the cache is not sufficiently large, after the execution of the first loop, only some elements that

*This work is supported in part by NSF Career Award #0093082 and by a grant from PDG.

CASES 2002, October 8–11, 2002, Grenoble, France.

are from the last part of the array will remain in the cache. The second loop, however, does not access these elements immediately. Instead, it starts with the first part of the array which can displace the last part of the array from the cache. When the second loop starts to access the last part, it is too late as these elements are not in the cache anymore. It should be emphasized that these two loops have a significant amount of data reuse; in fact, they share all elements of U. This means they have perfect inter-nest data *reuse*. However, inter-nest data *locality* is not good as we have explained above. As will be discussed in detail later in this paper, one solution in such a case is to run the second loop backwards (if data dependences allow this). If this can be done, then this loop (when it starts execution) will immediately access the last part of the array, increasing the hit rate. In this simple case, it is relatively easy to determine the best transformation. However, in general, when we have loop nests sharing multiple multi-dimensional arrays (that represent image/video data), we need a well-defined optimization algorithm to optimize inter-nest locality.

In this paper, we present an inter-nest data locality optimization algorithm and evaluate its effectiveness using a set of array-intensive applications from embedded image and video processing domain. Our algorithm does not only improve inter-nest data locality, but also ensures intra-nest locality after the code transformation. Our experimental results clearly show that the proposed approach is very successful in optimizing inter-nest locality. To the best of our knowledge, there exist three important previous studies on this topic. First, Catthoor et al. [3] present elegant techniques for making efficient use of limited data (memory) space considering access patterns exhibited by multiple nests. As compared to our work, their study is oriented more towards reducing the memory space requirements. Second, Ahmed et al. [1] propose a compiler strategy where multiple iteration spaces are first mapped into a common (large) iteration space and then this common iteration space is optimized for data reuse. Since not every pair of nests share arrays such an approach can potentially have a significant overhead. Also, reducing the dimensionality of the common iteration space, and finding a global transformation matrix for it are not easy tasks. Our experience indicates that most of inter-nest data reuse in array-intensive embedded applications occur between neighboring nests, motivating a technique that focuses on neighboring nests. Third, in an experimental study, McKinley and Temam [8] evaluate intra-nest and inter-nest data reuse in scientific applications, and find that an overwhelming majority of data reuse is intra-nest. This is in contrast to our study of embedded applications where there is significant inter-nest data reuse. This is because in many array-intensive embedded image/video applications the same data (e.g., an image) is processed by multiple nests one after another; this leads to a high degree of data reuse among nests. This is a strong motivation for the compiler-based strategy presented in this paper.

The rest of this paper is organized as follows. In Section 2, we define inter-nest locality formally and present a linear algebraic representation that can be used for describing/enhancing it by an optimizing compiler. In Section 3, we show how loop transformations can be used for inter-nest data locality. In Section 4, we present experimental results obtained through our implementation. Finally, in Section 5, we summarize our major contributions and discuss future work.

2. INTER-NEST LOCALITY

In this section, we present a linear algebraic representation that tries to capture inter-nest data reuse. Informally, this representation specifies how the arrays are accessed in a given loop nest. This information combined with the access pattern of the next loop nest enables our compiler to perform loop transformations on the latter. Such transformations are discussed in Section 3 in detail.

2.1 Background on Access Representation and Loop Transformation

The loops in a C program surrounding any statement can collectively be represented using an n-entry column vector (called iteration vector):

$$\vec{I} = (i_1, i_2, \cdots, i_n)^T,$$

where n is the enclosing loops (from outermost to innermost). Here, i_k is the kth loop index. The loop range or affine bounds of these loops can be described by a system of inequalities which define the polyhedron $\mathcal{S}\vec{I} \leq \vec{s}$, where $\mathcal{S}$ is a $k \times n$ matrix and $\vec{s}$ is a k-entry vector. If all loop bounds are constants, then we have $k = 2n$ (one upper and one lower bound per loop). The integer values that can be taken on by $\vec{I}$ define the iteration space of the nest. In a similar fashion, data (memory) layout of an array can also be represented using a polyhedron. This rectilinear polyhedron, called index space, is delimited by array bounds, and each integer point in it, called an array index, is represented using an index vector:

$$\vec{v} = (v_1, v_2, \cdots, v_m)^T,$$

where m is the number of dimensions of the array.

Based on these iteration space and index space (data space) definitions, an array access can be defined as a mapping from iteration space to index space, and can be described as:

$$\mathcal{G}\vec{I} + \vec{o}.$$

Assuming a nest with n loops that accesses an array of m dimensions, in this formulation, $\vec{I}$ denotes the iteration vector, $\mathcal{G}$ is an $m \times n$ matrix (called the access matrix) and $\vec{o}$ is a constant vector (called the offset vector). As an example, in a loop nest with two loops (i_1 and i_2), array reference $U[i_1 + 2][i_1 + i_2 - 3]$ can be represented as:

$$\mathcal{G}\vec{I} + \vec{o} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + \begin{pmatrix} 2 \\ -3 \end{pmatrix}.$$

The application of a loop transformation represented by a square, non-singular matrix $\mathcal{T}$ can be accomplished in two steps [11]: (i) re-writing loop body and (ii) re-writing loop bounds. For the first step, assuming that $\vec{I}$ is the vector that contains the original loop indices and $\vec{I'} = \mathcal{T}\vec{I}$ is the vector that contains the new (transformed) loop indices, each occurrence of $\vec{I}$ in the loop body is replaced by $\mathcal{T}^{-1}\vec{I'}$ (note that $\mathcal{T}$ is invertible). In other words, an array reference represented by $\mathcal{G}\vec{I} + \vec{o}$ is transformed to $\mathcal{G}\mathcal{T}^{-1}\vec{I'} + \vec{o}$. Determining the new loop bounds, however, is more complicated and, in general, may require the use of Fourier–Motzkin elimination (a method for solving an affine system of inequalities [11]). As an example, a loop nest that consists of i_1 (the outer loop) and i_2 (the inner loop) can be transformed into one with i_2 being the outer and i_1 being the inner using the following loop transformations matrix:

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

It should also be mentioned that any loop transformation should preserve the data dependences in the nest. If $\mathcal{D}$ is the data dependence matrix (i.e., a matrix whose each column is a dependence vector), after the loop transformation represented by $\mathcal{T}$, all columns of $\mathcal{T}\mathcal{D}$ should be lexicographically non-negative [11].

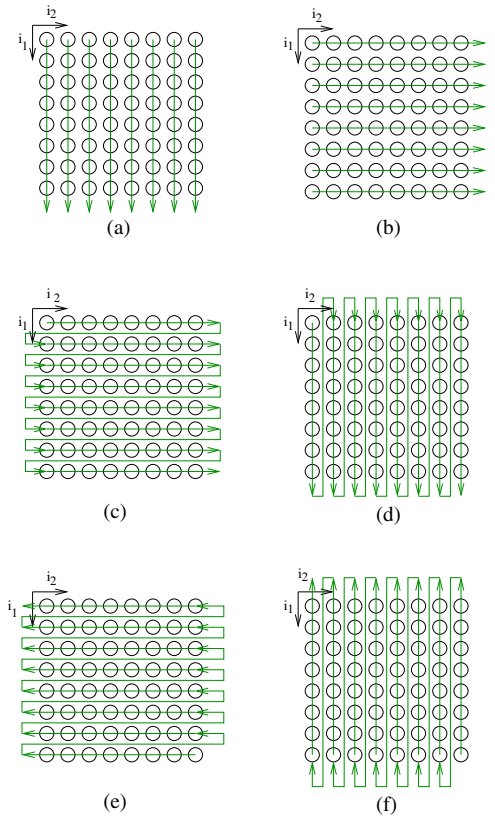

Figure 1: (a-b) Access directions along i_1 and i_2 loops. (c-d) Two different access patterns depending on the nesting order of the loops. (e-f) Access patterns in the opposite direction of (c-d).

2.2 Cache Footprint

When a loop nest finishes its execution, the array elements (belonging potentially to different arrays) left in the data cache constitute the *cache footprint* of the nest. In general, due to many factors involved (e.g., cache topology, access pattern, array base addresses), it is not possible to determine (statically) exactly which array elements remain in the cache after the execution of the nest. Therefore, what we need is a reasonably accurate estimation heuristic.

In search for such a heuristic, we note that the elements that will remain in the cache (i.e., cache footprint) depend strongly on how each array is traversed by the loop. Suppose, for example, that an array is traversed one row at a time from top to bottom. Consequently, when the nest finishes execution, we can expect that, the elements that will remain in the cache will be the ones that belong to the last rows visited (the exact number of these rows depends on the cache capacity and array base addresses). Based on this, we can argue that the next loop nest should be transformed (optimized) in such a way that the first elements that it will be accessing from the array in question should be the ones that belong to these rows. Therefore, a first-order approximation for cache footprint would be capturing the direction of accesses. In the next subsection, we propose a mathematical representation, called footprint vectors, that can be used for this purpose.

2.3 Footprint Vectors

In order to restructure a given loop nest taking into account the access pattern of the previous nest (and, thus, the contents of the cache), we need a representation that summarizes the access pattern of this (previous) nest. There are three major requirements that need to be satisfied by such a representation. First, it should summarize the access pattern as accurately as possible. Second, it should be concise enough so that it can be manipulated easily (e.g., using linear algebra). Third, it should be independent of the cache architecture used. This last issue is important from the modularity viewpoint. This is because most loop/data optimizations are applied during the initial phases of compilation, and these phases generally do not exploit too many architecture-specific characteristics.

To capture data access pattern of a given array reference in a loop nest, we use a representation called *footprint vector*. A footprint vector corresponds to the access direction by a given reference on the array index space. Let $\mathcal{G}\vec{I} + \vec{o}$ be a reference to an m-dimensional array in a nest of n loops. We define two iteration vectors: $\vec{I_p}$ and $\vec{I_p'}$ such that $\vec{I_p'} - \vec{I_p} = (0, 0, \cdots, 0, 0, 1, 0, 0, \cdots, 0, 0)^T$; that is, $\vec{I_p}$ and $\vec{I_p'}$ are two loop iterations that only differ by 1 in loop position p, where $1 \leq p \leq n$. Note that we also implicitly assume that when we move from iteration $\vec{I_p}$ to iteration $\vec{I_p'}$, we do not cross the loop upper bound.

A footprint vector with respect to loop position p is defined as the difference between the two array elements accessed by $\vec{I_p'}$ and $\vec{I_p}$:

$$\vec{f_p} = (\mathcal{G}\vec{I_p'} + \vec{o}) - (\mathcal{G}\vec{I_p} + \vec{o}) = \mathcal{G}(\vec{I_p'} - \vec{I_p}) = \vec{g_p}.$$

Here, $\vec{g_p}$ is the pth column of $\mathcal{G}$, the access matrix. It should be noted that a footprint vector is defined with respect to an array reference and a loop position. Therefore, a given array reference, we can have multiple footprint vectors, one for each loop position. For example, if:

$$\mathcal{G}\vec{I} + \vec{o} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} + \begin{pmatrix} 2 \\ -3 \end{pmatrix},$$

then we have:

$$\vec{g_1} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \text{and} \quad \vec{g_2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Note also that, in general, there might be multiple references to a given array in the same loop nest. Consequently, for each loop position, we can define a matrix, each column of which is a footprint vector. For example, if we have two references $U[i_1][i_2 + 1]$ and $U[i_2][i_1 + i_2]$ to an array U in a nest with loops i_1 (the outer loop) and i_2 (the inner loop), we have two such matrices, one for loop index i_1 and the other one for loop index i_2:

$$F_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad F_2 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Let us now discuss the meaning of a footprint vector. A footprint vector (defined with respect to an array reference and loop position) indicates how (i.e., in which direction) the array in question is traversed when the iterations of the corresponding loop are executed. For example, a footprint vector with respect to an innermost loop position indicates how this loop traverses the array in question. Consequently, when the nest finishes its execution, we might have some idea about the array elements that might be in the cache. Then, the next nest can be transformed to take advantage of these elements in the cache. For instance, let us focus on the following two footprint vectors:

$$\vec{g_1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad \vec{g_2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Here, we assume that $\vec{g_1}$ corresponds to i_1 loop and $\vec{g_2}$ corresponds to i_2 loop. Figures 1(a) and (b) illustrate how an array (with these footprint vectors) is traversed in i_1 and i_2 directions, respectively. Note that these directions correspond to the associated footprint vectors. The overall access pattern, on the other hand, depends on the nesting order of these two loops. Assuming the i_1 loop is the outer loop, we have the access pattern shown in Figure 1(c). On the other hand, if we assume that loop i_2 is the outer loop, we obtain the access pattern illustrated in Figure 1(d). If we look at Figure 1(c) (resp. Figure 1(d)) more closely, we can observe that, after the execution of the nest, we can expect that the cache will contain the last rows (resp. the last columns) of the array. Note that the footprint vectors (in conjunction with nesting order) give us this information. Therefore, they can be used in optimizing the nest that follows (in execution) this nest.

An important problem that needs to be solved now is the question of how to decide the access direction when we have multiple references to the same array. This is because each reference might impose a different access pattern, requiring a different footprint vector to represent it. While it might be possible to record all these footprint vectors and use them later in optimizing the next nest, if there are too many references to the same array, it might be a better idea to try to *unify* these vectors into a single vector. Note that this problem can be re-expressed as one of generating a single footprint vector from multiple footprint vectors. While one can use several operators to unify multiple vectors, in this study, we use vector combination.[1] Note that, in a sense, the combined vector reflects (or represents) the access pattern of each vector involved. In our context, it can be used to represent the overall access pattern of exhibited by multiple references to the same array. In the remainder of this paper, unless stated otherwise, when we mention footprint vector for an array and a loop position, we mean the combined footprint vector (that is, the contributions of all references are taken into account). However, it should be stressed that we still need multiple footprint vectors for each array as we need to represent the array traversal in each loop direction.

3. INTER-NEST OPTIMIZATION

Once the footprint vectors of the previous nest are captured, the next task in our optimization strategy is to transform the current nest based on these footprint vectors. In this section, we focus on this problem and present a loop transformation framework for enhancing inter-nest data locality.

3.1 Determining Loop Transformations

We start by observing that in order to take best advantage of the data left in the cache after the execution of the previous nest, the array should be traversed in opposite direction (as compared to the traversal direction in the first nest) in all array dimensions. Consider, for example, the access pattern shown in Figure 1(c). Assuming that this is the access pattern exhibited by the previous nest, for the best inter-nest data locality, the current nest should generate the access pattern illustrated in Figure 1(e). Note that this pattern corresponds to the following footprint vectors:

$$\vec{g_1'} = \begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad \text{and} \quad \vec{g_2'} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}.$$

So, the current nest should be transformed such that these footprint vectors are obtained. Similarly, if the access pattern of the previous nest is the one shown in Figure 1(d), the access pattern of the second nest should be the one given in Figure 1(f). The following lemma gives our major result.

Lemma 1. Let $\mathcal{H}$ be a matrix such that column k is the footprint vector k of the "previous" nest (i.e., the footprint vector that corresponds to the kth loop position). Such a matrix is referred to as the *footprint matrix*. If the loop transformation matrix $\mathcal{T}$ (for the "current" nest) satisfies $\mathcal{N}\mathcal{T}^{-1} = \mathcal{H}'$, (where $\mathcal{N}$ is the default footprint matrix of the current nest and $\mathcal{H}'$ is $\mathcal{H}$ with all entries negated), the inter-nest data locality is improved.

Sketch of the Proof. Since $\mathcal{H}$ represents the footprint vectors in the previous nest, $\mathcal{H}'$ should contain the target footprint vectors for the current nest. We also note that the columns of $\mathcal{H}$ and $\mathcal{N}$ correspond to the columns of the respective access matrices. Consequently, when transforming the nest using the loop transformation matrix $\mathcal{T}$, these columns should be transformed the same way that the access matrices are transformed; so, we need to select a $\mathcal{T}$ such that $\mathcal{N}\mathcal{T}^{-1} = \mathcal{H}'$ is satisfied.

It should be noted, however, that the loop transformation determined based on $\mathcal{N}\mathcal{T}^{-1} = \mathcal{H}'$ may not be able to generate the optimal transformation in absolute sense; it is just expected to give us an improved version. As an example, let us consider the following program fragment which consists of two separate nests:

```
for (i1 = 0; i2 < n; i1++)
  for (i2 = 0; i2 < n; i2++)
    { ...U[i1][i2]... }
for (i3 = 1; i3 < n; i3++)
  for (i4 = 0; i4 < n - 1; i4++)
    { ...U[i3 - 1][i4 + 1]... }
```

Considering the original access patterns of these nests, we have:

$$\mathcal{H} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad \mathcal{N} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Consequently, the second nest should be transformed using a loop transformation $\mathcal{T}$ such that:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathcal{T}^{-1} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$$

should be satisfied. So, we have the following transformation matrix for the second nest:

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

Applying the loop transformation represented by this $\mathcal{T}$, we obtain:

```
for (i3 = n; i3 > 1; i3--)
  for (i4 = n - 1; i4 > 0; i4--)
    { ...U[i3 - 1][i4 + 1]... }
```

It should be observed that this transformed loop tries to reuse the data left in the cache by the first nest as much as possible. However, in general, there might be multiple arrays accessed in the loops. Since these arrays can have different footprint matrices, the selection of the loop transformation matrix (for the second nest) should be made by considering all footprint matrices. For example, in the program fragment above, if we have a reference to an array V (in addition to that to array U) in each nest, then we need to determine a transformation matrix $\mathcal{T}$ such that both $\mathcal{N}_u\mathcal{T}^{-1} = \mathcal{H}'_u$ and $\mathcal{N}_v\mathcal{T}^{-1} = \mathcal{H}'_v$ are satisfied. Here, $\mathcal{H}'_u$ and $\mathcal{H}'_v$ are the opposite (i.e., the negated versions) of the original footprint matrices $\mathcal{H}_u$ and $\mathcal{H}_v$ that contain the footprint vectors for array U and array V, respectively, in the first nest. $\mathcal{N}_u$ and $\mathcal{N}_v$, on the other hand, represent the footprint matrices that hold the original footprint vectors in the second nest. In some cases, especially when there are

[1] A linear combination of the vectors $\vec{x}$, $\vec{y}$, and $\vec{z}$ is given by $a\vec{x} + b\vec{y} + c\vec{z}$, where a, b, and c are constants.

too many arrays shared by multiple nests, it may not be possible to find a $\mathcal{T}$ in processing a given loop nest. In such cases, one solution is to drop some arrays from consideration and repeat the process of determining a suitable $\mathcal{T}$. While there may be many different strategies to determine the array(s) to be dropped, in our current implementation, we use profile data and select the array that is used the least frequently in the program. More specifically, we instrument the program to be optimized so that at runtime it keeps track of how many times each array reference is touched. Then, by considering all the references to each array, we detect the array which is referenced the least frequently. This is the array dropped from consideration before we try to solve the system again.

3.2 Integrating with Intra-Nest Optimizer

While the technique discussed so far improves inter-nest data locality, a loop transformation determined based on inter-nest reuse (between the previous and current nests) alone might hurt the intra-nest locality in the current nest. Consequently, in selecting the loop transformation (for the current nest), we need also consider intra-nest data reuse. In this subsection, we address this issue.

Many locality-oriented studies use the *reuse vector* concept for representing and optimizing intra-nest data locality. The data reuse theory introduced by Wolf and Lam [10] and later refined by Li [7] can be used for identifying the types of reuses in a given loop nest. Two iterations represented by vectors $\bar{I}'$ and $\bar{I}''$ (where $\bar{I}'$ precedes $\bar{I}''$ in original execution) access the same array element using the reference represented as $\mathcal{G}\bar{I}+\bar{o}$ if $\mathcal{G}\bar{I}'+\bar{o}=\mathcal{G}\bar{I}''+\bar{o}$. In this case, the *temporal reuse vector* is defined as $\bar{r}=\bar{I}''-\bar{I}'$, and it can be computed from the relation $\mathcal{G}\bar{r}=\bar{0}$. Assuming row-major memory layouts for multi-dimensional arrays (as in C), spatial reuse occurs if the accesses are made to the same row. We can compute the *spatial reuse vector* $\bar{r'}$ from the equation $\mathcal{G}_s\bar{r'}=\bar{0}$, where $\mathcal{G}_s$ is $\mathcal{G}$ with all elements of the last row replaced by zero [10, 7]. A collection of individual reuse vectors (originating potentially from different arrays and different references) is referred to as a *reuse matrix*, $\mathcal{R}$. It is also known [7] that a loop transformation represented by $\mathcal{T}$ transforms a reuse matrix $\mathcal{R}$ to $\mathcal{T}\mathcal{R}$. Consequently, a loop transformation matrix that is acceptable from both inter-nest and intra-nest data reuse perspectives should satisfy both $\mathcal{N}\mathcal{T}^{-1}=\mathcal{H}'$ and $\mathcal{T}\mathcal{R}=\mathcal{R}'$, where $\mathcal{R}'$ is the desired reuse matrix and $\mathcal{N}$ and $\mathcal{H}'$ are as defined in Lemma 1. Li's thesis [7] explains a strategy to determine a suitable $\mathcal{R}'$ matrix. In summary, in the ideal case, each column of this matrix should have its non-zero element as the last entry (all other entries being 0). This is because such a reuse vector implies that data reuse occurs between the iterations of the innermost loop; thus, the chances are high that it can be converted into locality (at runtime). Since in general a reuse matrix can contain multiple vectors, it may not be possible to bring each column into this form. The approach proposed by Li first orders the vectors (in the reuse matrix) from left to right according to non-increasing frequency of occurrence; that is, if a reuse vector occurs more frequently than another one, it is placed to the left of the latter. After that, the approach tries to achieve the ideal reuse vector for as many columns as possible from the left side of the matrix. The details of this approach are beyond the scope of this paper and can be found in [7].

For example, suppose that the original second nest in the code fragment above was:

$$\begin{array}{l}\text{for } (i_3=0; i_3<n-1; i_3++)\\ \quad\text{for } (i_4=1; i_4<n; i_4++)\\ \quad\quad\{\ ...U[i_4-1][i_3+1]...\ \}\end{array}$$

In this case, we have only spatial reuse which is carried by the outer loop. This is because for a fixed value of the inner loop, the successive iterations of the outer loop access the consecutive array elements. So, the reuse matrix is

$$\mathcal{R}=\begin{pmatrix}1\\0\end{pmatrix}.$$

Then, we need to select a transformation matrix $\mathcal{T}$ such that both of the following equalities are satisfied:

$$\begin{pmatrix}0&1\\1&0\end{pmatrix}\mathcal{T}^{-1}=\begin{pmatrix}-1&0\\0&-1\end{pmatrix}\quad\text{and}\quad\mathcal{T}\begin{pmatrix}1\\0\end{pmatrix}=\begin{pmatrix}0\\\mp1\end{pmatrix}.$$

Note that the first equality here corresponds to inter-nest constraint, whereas the second one corresponds to intra-nest constraint. Also, the vector of the right hand side of the second equality refers to the ideal spatial reuse vector (matrix) for a nest with two loops [7]. A loop transformation matrix that satisfies both these constraints is

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

which results in the following code (i.e., the transformed version of the second nest):

$$\begin{array}{l}\text{for } (i_3=n; i_3>1; i_3--)\\ \quad\text{for } (i_4=n-1; i_4>0; i_4--)\\ \quad\quad\{\ ...U[i_3-1][i_4+1]...\ \}\end{array}$$

Note that this nest is acceptable from both intra-nest locality and inter-nest locality perspectives.

3.3 Considering Multiple Previous Nests

While so far we have considered only the inter-nest data reuse between two neighboring nests, in some cases, inter-nest reuse can exist (and may be exploited) between two nests that are not neighbors. For example, two nests that have another one between them can access the same set of array elements while the nest between them does not access the said array elements. In such a case, it may or may not be beneficial to transform the third nest (the one that reuses the data) to exploit inter-nest reuse. This decision is largely dependent on whether the intermediate nest allows any data (in the cache) processed by the first nest to reach to the third nest. We check whether this is really the case using the Omaga Library [6]. The Omega Library is a set of routines for manipulating linear constraints over integer variables, Presburger formulas, and integer tuple relations and sets. By using this library, we can determine how many distinct elements have been accessed in the intermediate nest. If this number is (at least) close to the cache capacity, there is not much point in trying to optimize the inter-nest data reuse between the first and the third nests. If, on the other hand, this number is significantly smaller than cache capacity, there might be some benefit in transforming the third nest for locality. If this is the case, we use the approach explained in this paper to optimize the data reuse between these nests.

3.4 Overall Optimization Strategy

Our overall optimization strategy is given in Figure 2. For clarity of presentation, we assume that in a given nest each array is accessed using a single reference. In this algorithm, $f(j)$ is the number of arrays accessed in nest $\mathcal{N}_j$. $\mathcal{H}_{k,j}$ represents the footprint matrix for array j in nest $\mathcal{N}_k$, and $\mathcal{R}_k$ and $\mathcal{T}_k$ are the reuse matrix and the loop transformation matrix, respectively, for nest $\mathcal{N}_k$. $\mathcal{H}'_{k,j}$ is the same as $\mathcal{H}_{k,j}$ except that all entries are negated, and $\mathcal{R}'_k$ is the desired (target) reuse matrix for nest $\mathcal{N}_k$. The first nest in the code is optimized for only intra-nest reuse (between lines 1 and 4)

1. *Determine* $\mathcal{H}_{1,i}$ *for each* $1 \leq i \leq f(1)$
2. *Determine* $\mathcal{R}_1$
3. *Solve* $\mathcal{T}_1 \mathcal{R}_1 = \mathcal{R}'_1$ *for a suitable* $\mathcal{T}_1$
4. *Transform nest* $\mathcal{N}_1$ *using* $\mathcal{T}_1$
5. *For k = 2, last nest*
6. *Determine set S, the set of common arrays between* $\mathcal{N}_k$ *and* $\mathcal{N}_{k-1}$
7. *For each* $s \in S$ *do*
8. *Build* $\mathcal{H}_{k,s} {\mathcal{T}_k}^{-1} = \mathcal{H}'_{k-1,s}$
9. *Endfor*
10. *Determine* $\mathcal{R}_k$
11. *Build* $\mathcal{T}_k \mathcal{R}_k = \mathcal{R}'_k$
12. *Determine a suitable* $\mathcal{T}_k$ *from constraints in (8) and (11)*
13. *Transform nest* $\mathcal{N}_k$ *using* $\mathcal{T}_k$
14. *Endfor*

Figure 2: Locality optimization algorithm.

and the remaining nests are optimized for both intra-nest reuse and inter-nest reuse (in the loop between lines 5 and 14). Note that, in determining the loop transformation matrix $\mathcal{T}_k$, we use both inter-nest constraint (i.e., $\mathcal{H}_{k,s} {\mathcal{T}_k}^{-1} = \mathcal{H}'_{k-1,s}$) and intra-nest constraint (i.e., $\mathcal{T}_k \mathcal{R}_k = \mathcal{R}'_k$). The target reuse matrix $\mathcal{R}'_k$ can be determined using the strategy proposed by Li [7]. Also, as mentioned earlier in the paper, in determining a loop transformation matrix, when we have conflicts, we drop one or more arrays from consideration. This is not shown explicitly in the algorithm for the sake of clarity. Finally, once a loop transformation matrix $\mathcal{T}_k$ is determined for nest $\mathcal{N}_k$, we also check whether all the columns in $\mathcal{T}_k \mathcal{D}_k$ are lexicographically non-negative (where $\mathcal{D}_k$ is the data dependence matrix). If not, then the transformation is not applied.

4. EXPERIMENTS

Our inter-nest data locality optimization algorithm is implemented using the SUIF experimental compiler infrastructure from Stanford University [2]. In order to test the effectiveness of our approach, we used four array-intensive applications from the image processing domain, available to us from UMIST. `wave` is a wavelet compression code that targets specifically medical applications. This code has a characteristic that it can reduce image data to an extremely small fraction of its original size without compromising image quality significantly. `splat` is a volume rendering application which is used in multi-resolution volume visualization through hierarchical wavelet splatting. It is used primarily in the area of morphological image processing. `3D` is an image-based modeling application that simplifies the task of building 3D models and scenes. Finally, `dfe` is a digital image filtering and enhancement code. It also contains a module for shape detection. These C programs are written so that they can operate on images of different sizes. The input sizes used in our experiments vary between 117KB and 231KB, and the execution times of the original codes range from 21.2 seconds to 118.6 seconds. All the results presented in this section are obtained using an in-house execution-driven simulator that simulates an embedded MIPS processor core (a MIPS 4Kp like architecture). The simulated architecture has a five-stage pipeline that supports four execution units (integer, multiply-divide, branch control, and processor control). This core is operated at 200MHz frequency, has thirty-two, 32-bit general-purpose registers, and includes 8KB instruction cache and 8KB data cache. The cache hit latency is 2 cycles and the miss latency is 75 cycles. The simulator takes a C code as input, simulates its execution, and produces statistics including hit/miss behavior and execution time.

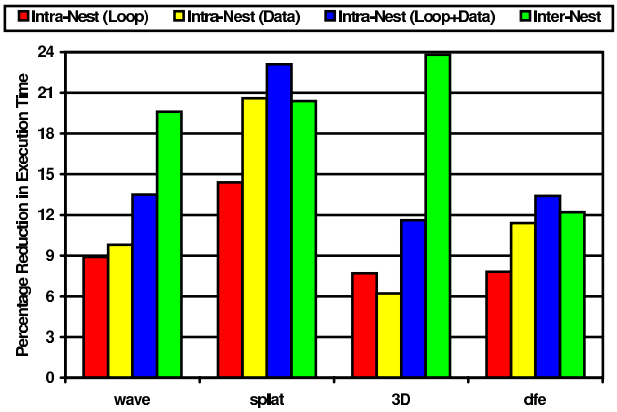

Figure 3: Percentage improvement in execution time (with respect to Original).

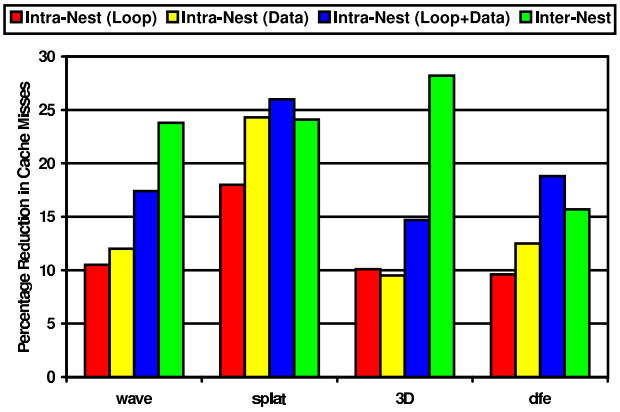

Figure 4: Percentage improvement in data cache misses (with respect to Original).

Using SUIF, we generated five different versions of each code in our experimental suite:

- *Original:* This is the original code.
- *Intra-Nest (Loop):* This is a pure loop-oriented data locality optimization strategy. It optimizes a given code nest-by-nest using popular loop transformations including loop interchange, loop reversal, loop skewing, loop scaling, and tiling. Details of this approach can be found in Li's thesis [7].
- *Intra-Nest (Data):* This is a pure data layout optimization strategy. It does not transform loop structures; instead, it enhances locality by re-layouting data in memory based on the access patterns exhibited by the application code. Details of this strategy can be found in [9].
- *Intra-Nest (Loop+Data):* This is a data locality optimization approach based iteration and data space transformations. Specifically, for each loop nest, it first uses aggressive loop transformations to optimize temporal locality; and then, for array references without temporal locality, it uses data transformations (layout modifications) for optimizing their spatial locality. A detailed description of this approach is outside the scope of this paper and can be found elsewhere [5].
- *Inter-Nest:* This is the optimization strategy presented in this paper. It tries to capture inter-nest reuse as much as possible. But, as explained earlier, in selecting the loop transformation, it also takes intra-nest data reuse into account.

It should be mentioned that Intra-Nest (Loop), Intra-Nest (Data), and Intra-Nest (Loop+Data) represent the state-of-the-art for pure loop, pure data, and combined loop and data optimizations, respectively. For the codes in our experimental suite, using our inter-nest optimization scheme increased the overall compilation times by at most 40% (as compared to the compilation times of the original codes). Considering the runtime benefits of our optimization scheme, this increase in compilation time is tolerable. Particularly, in many embedded designs, a large number of machine cycles can be spent in compilation as these systems typically run a single application or a small set of related applications, and the increases in compilation times can be compensated for by the large quantities of end-products shipped. All percentage improvements presented in this section are with respect to Original.

Figure 3 presents the percentage improvements in execution times when different strategies are used. We see that the average execution time improvements due to Intra-Nest (Loop), Intra-Nest (Data), Intra-Nest (Loop+Data), and Inter-Nest versions are 9.7%, 12%, 15.4%, and 19%, respectively. We observe that our approach is always more successful than Intra-Nest (Loop); that is, optimizing inter-nest locality can bring large performance benefits. We also observe that in two of our four applications (`wave` and `3D`) our approach even generates better results than Intra-Nest (Loop+Data). This is due to significant amount of inter-nest data reuse in these two applications. In other two applications, on the other hand, the Intra-Nest (Loop+Data) version generates better results than ours. This is because the former also employs data space transformations. To see where these execution time benefits are coming from, we show in Figure 4 the percentage reductions in data cache misses (over the Original version). We see from these results that the trends in cache miss savings are very similar to those in execution time savings.

Recall that the Inter-Nest version takes both inter-nest and intra-nest data reuse into account in a selecting loop transformation matrices. To see whether just considering inter-nest constraints would

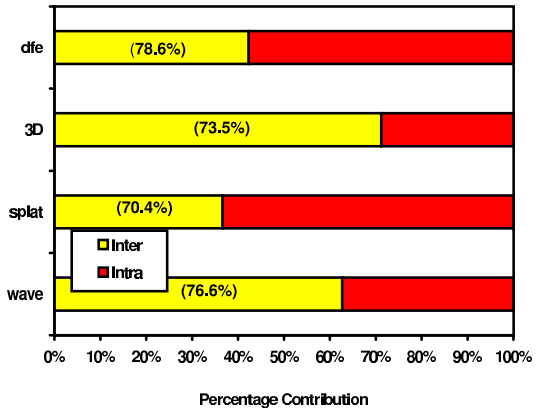

Figure 5: Contributions of inter-nest reuse and intra-nest reuse to the savings of the Inter-Nest version.

be sufficient, we present in Figure 5 the contribution of inter-nest constraints and intra-nest constraints to the execution time savings. In particular, for each application, the left portion of the bar (in Figure 5) shows the percentage of benefits that are due to just optimizing inter-nest data locality. We clearly see from these results that optimizing just inter-nest locality is not sufficient; there is a significant amount of intra-nest data reuse that needs to be accounted for. The numbers written inside the left portions (of the bars), on the other hand, indicate the percentage of potential data reuse that has really been exploited. For example, in splat, 70.4% of the available inter-nest reuse has been exploited. The maximum potential inter-nest data reuse (between two neighboring nests) has been calculated by intersecting the following two sets and finding the number of elements in the resulting set: the set of array elements that remain in the cache after the execution of the first nest and the set of array elements that are required by the second nest. This process is repeated for each neighboring nest pair and the resulting total sum represents the maximum inter-nest data reuse in the program (we refer to this number as MDR). On the other hand, the inter-nest data reuse that has really been exploited between two neighboring nests has been calculated by intersecting these two sets and finding the number of elements in the resulting set: the set of array elements that remain in the cache after the execution of the first nest and the set of array elements that are reused from the cache by the second nest. As before, this process is repeated for each nest neighboring nest pair. The resulting total sum (over all nest pairs) represents the inter-nest data reuse that has actually been exploited (we refer to this number as EDR). The numbers written within the black portions of the bars in Figure 5 correspond to EDR/MDR. We see that at least 70% of all inter-nest data reuse are converted into locality. However, while these numbers are encouraging, they also indicate that there is even opportunity for achieving better inter-nest locality.

A potential source for our inability to exploit inter-nest locality fully is the conflicting requirements that the compiler needs to resolve for determining the loop transformation matrix ($\mathcal{T}$) for a given nest. Such conflicts may occur when different arrays in a given nest demand different loop transformation matrices. As mentioned earlier, our solution to this is to drop an array from consideration and try to find the transformation matrix again. This process (eliminating an array) continues until we find a $\mathcal{T}$ without conflict. Obviously, a greater number of arrays accessed in a loop will increase the chances of a conflict. The graph in Figure 6 gives, for each benchmark, the percentage of conflicts when we have 2, 3, 4,

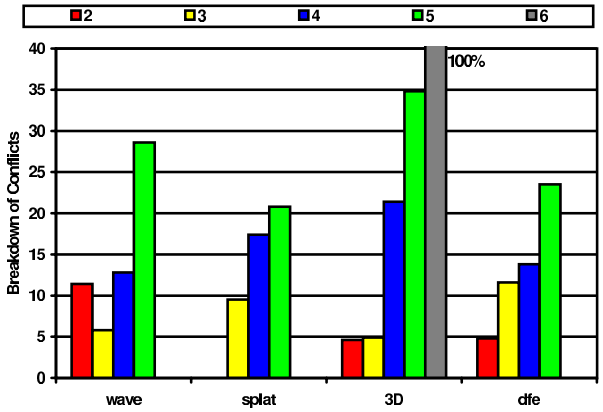

Figure 6: Percentage of conflicts in determining $\mathcal{T}$ as a function of the number of arrays accessed in each nest.

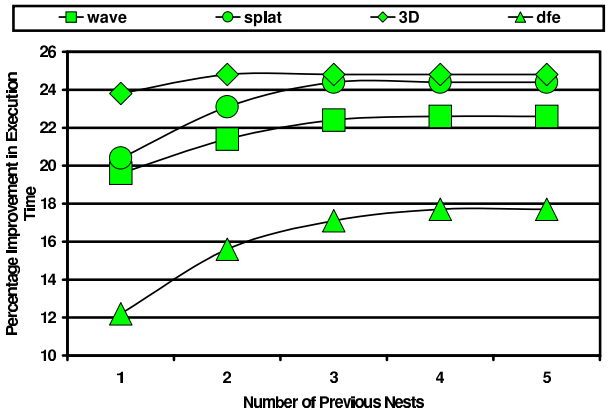

Figure 7: Increasing the scope of inter-nest reuse.

5, 6 arrays in the nest. For example, in `wave`, out of the nests that access two arrays, only 11.4% present a conflict in determining the loop transformation matrix. As expected, in general, when we have more arrays in the nest being optimized, the number of conflicts is higher. Nevertheless, these numbers are not too high indicating that our inter-nest optimizer does not suffer from too many conflicts. This is because in a given nest that accesses multiple arrays, these arrays are in general accessed using the same access matrix. Therefore, a loop transformation which is desirable for one array is desirable for others too.

Our baseline optimization strategy only focuses on the inter-nest data reuse between two neighboring nests. In some cases, however, there might be data reuse between nests that are not neighbors. To see whether this occurs for our benchmarks, we performed another set of experiments where we changed the number of previous nests considering for optimizing the inter-nest reuse in a given loop nest (using the approach discussed in Section 3.3). Each point in the x-axis in Figure 7 corresponds to a different value for this number. Note that the original value of this number is 1. We observe from these results that increasing the number of previous nests considered makes some difference in some cases; but, when we move beyond three previous nests, the additional benefits are incremental.

Figure 8 demonstrates how performance savings are affected when cache size is changed (from 2K to 32K). These results are average savings across all four benchmarks in our experimental suite. We see that our approach is most effective with small cache sizes (which are more common in embedded systems). However, even with a 16K data cache, it outperforms the remaining strategies. The reason that the percentage savings due to our strategy reduce when the cache size is increased can be explained as follows. When the cache capacity is increased, the Original version starts to perform better as more data are captured in the cache (as compared to the case with the small cache size). While our approach also performs better with the increased cache size, the improvement rate of the original codes is much higher than that of the optimized codes. Consequently, we observe a reduction in savings as we increase the cache capacity.

5. CONCLUSIONS AND FUTURE WORK

As incresingly large portions of functionality of embedded systems are being implemented in software, many of these systems offer caches to speed up computations. While most compiler-based optimization techniques currently restructure code for cache locality from the viewpoint of intra-nest locality, our experience with several array-intensive embedded applications clearly show that there also exists a significant amount of inter-nest data reuse. Based on this observation, we proposed in this paper a compiler algorithm for exploiting inter-nest data reuse. We also demonstrated how our technique can be combined with an intra-nest optimizer. Our experiments with four array-intensive embedded applications revealed that it is possible to obtain execution time savings ranging from 9.7% to 19%.

This work is a part of the ongoing effort at Penn State on enhancing performance of embedded applications. Currently, there are two major shortcomings of this work. First, the proposed algorithm operates on a procedure granularity, and therefore, can fail to fully-optimize some applications that can benefit from interprocedural analysis. Second, the current optimizer does not consider data (memory) layout transformations. Since footprint vectors are defined on data space, they can be manipulated (optimized) using data transfomations as well. Our future research will address both these issues.

6. REFERENCES

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proc. the International Conference on Supercomputing,* May 2000.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proc. the Seventh SIAM Conf. on Parallel Proc. for Scientific Computing,* February, 1995.

[3] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P.G. Kjeldsberg, T. V. Achteren, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors,* Kluwer Academic Publishers, 2002.

[4] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design.* Kluwer Academic Publishers, 1998.

[5] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proc. International Symposium on Microarchitecture,* Dallas, TX, December, 1998.

[6] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and

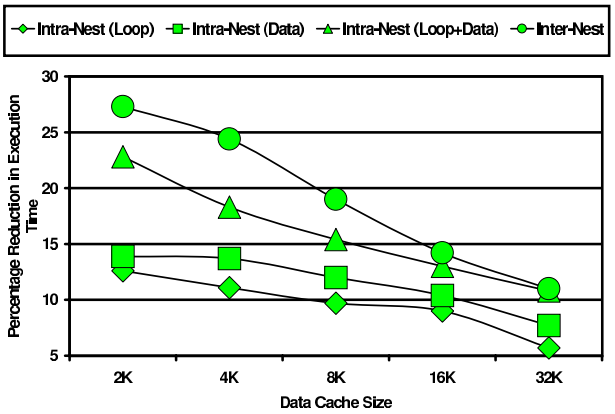

Figure 8: Percentage improvement (over Original) in execution time with different cache capacities.

David Wonnacott. The Omega Library interface guide. *Technical Report CS–TR–3445,* Computer Science Department, University of Maryland, College Park, MD, March 1995.

[7] W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, New York, 1993.

[8] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proc. the 7th International Conference on Architectural Support for Programming Languages and Operating Systems,* Boston, MA, October 1996.

[9] M. F. P. O'Boyle and P. M. W. Knijnenberg. Non-singular data transformations: definition, validity and application. In *Proc. International Conference on Supercomputing,* Vienna, July 1997.

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

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