Data Relation Vectors: A New Abstraction for Data Optimizations

Mahmut Kandemir
Dept. of Computer Science and Engineering
Pennsylvania State University
kandemir@cse.psu.edu

J. Ramanujam
Dept. of Electrical and Computer Engineering
Louisiana State University
jxr@ee.lsu.edu

Abstract

This paper presents an abstraction called *data relation vectors* to improve the data access characteristics and memory layouts in a given regular computation. The key idea is to define a relation between the data elements accessed by close-by iterations and use this relation to guide to a number of optimizations for array-based computations. The specific optimizations studied in this paper include enhancing group-spatial and self-spatial reuses, improving intra-tile and inter-tile reuses, and reducing unnecessary communication on message-passing architectures. In addition, this abstraction works well with other known abstractions such as data reuse vectors. The data relation vector abstraction has been implemented in the SUIF compilation framework and has been tested using a set of twelve benchmarks from image processing and scientific computation domains. Preliminary results on a super-scalar processor show that it is successful in reducing compilation time and outperforms two previously proposed techniques, one that uses only loop transformations and one that uses both loop and data transformations. Our experiments also show that the proposed abstraction helps one to select good data tile shapes which can subsequently be used to determine iteration space tiles.

1 Introduction

Several techniques have been proposed for expoiting locality in loop nests. These include (i) loop optimization strategies such as linear loop transformations, tiling, loop fusion, loop fission, and loop unrolling [31]; (ii) data transformations, which change the memory layout of data [20, 14]; and (iii) combined loop and data transformation techniques [8, 12, 13, 24]. These transformation techniques need to use some kind of abstraction (a) for expressing the inherent locality exhibited by the code, and (b) for deriving optimal (or enhanced) loop orders and/or memory layouts. So far, a majority of the approaches to optimizing data locality in structured scientific codes use some form of *reuse vectors,* i.e., vectors that express the *relationship between loop iterations* that access nearby memory locations (spatial proximity) [21, 29]. While data reuse vectors have proved to be excellent tools for manipulating iteration spaces (or loops) for improving data locality and/or parallelism, we believe that, for optimizing the memory layouts of multi-dimensional data structures, compilers need to use a different abstraction that works *directly* on the *data space* (array layout domain), in contrast to reuse vectors that work on iteration space. In this paper, we propose and evaluate the effectiveness of such a *data space-oriented abstraction* called *data relation vectors* (DRV) and show how it can be used for a variety of purposes within an optimizing compiler framework. Specifically, in this paper, we make the following contributions:

- We present a new abstraction, namely data relation vectors (DRV), that can be used for manipulating memory layouts of multi-dimensional arrays.
- We show how this new abstraction can be used with different optimization goals. The optimizations that can benefit from DRV range from spatial locality improvement for single processors to communication optimizations for parallel machines.
- We show how DRVs can be combined with data reuse vectors in a unified framework, and how these two abstractions can work together for improving data locality by manipulating loop and data spaces simultaneously. In particular, we show how group-spatial, self-spatial, and temporal reuse can be improved.
- We present preliminary results from our initial implementation in a source-to-source translator and compare these results with those obtained using other locality optimization techniques.

Overall, we believe that the impact of such an approach is two-fold. First, as far as manipulating memory layouts of arrays is concerned, DRVs are easier to obtain than reuse vectors (especially in codes that exhibit uniformly generated references [9]). In some cases, data transformations might be the only alternative for enhancing spatial locality; therefore, this paper presents an abstraction which makes application of data transformations easier. Second, combining DRV with other abstractions will open new research avenues for compiler writers.

The rest of this paper is organized as follows. Section 2 reviews the data reuse vector concept and show how it is used for locality optimization purposes. In Section 3, we introduce data relation vectors, establish their relationship with reuse vectors, and discuss how they can be used for a number of optimizations. In Section 4, we show how the DRV concept and reuse vector concept can be combined within a unified optimization framework. In Section 5, we present our experimental results. We summarize related work in Section 6 and we conclude the paper with a summary and outline of future work in Section 7.

2 Preliminaries

In this section, we first present the mathematical notation used in this paper to express loop iterations, array accesses, and affine references. Then, we review the reuse vector concept and explain how it is used for optimizing data locality.

Consider a reference to an m-dimensional array in a loop nest of depth n. Our scope of optimization is *regular scientific codes;* consequently, we assume that the array subscript functions and loop bounds are *affine functions* of enclosing loop indices and symbolic constants. For a given loop nest, we use $\bar{i}$ to denote the *iteration vector* (consisting of loop indices starting from the outermost loop to the innermost). So, each affine array reference can be represented as $H\bar{i} + \bar{c}$, where the $m \times n$ matrix H is called the *access* (or *reference*) *matrix* [29, 31] and the m-element vector $\bar{c}$ is referred to as the *offset vector* [29, 31].

The data reuse theory introduced by Wolf and Lam [29] and later refined by Li [21] can be used to identify 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 sequential execution) access the same data element using a reference represented as $H\bar{i} + \bar{c}$ if $H\bar{i'} + \bar{c} = H\bar{i''} + \bar{c}$. In this case, the *self-temporal reuse vector* is defined as $\bar{r} = \bar{i''} - \bar{i'}$, and it can be computed from the relation $H\bar{r} = \bar{0}$. Assuming *column-major* memory layouts for arrays, self-spatial reuse can occur if the accesses are made to the same column. We can compute the *self-spatial reuse vector* $\bar{r}$ from the equation $H_r\bar{r} = \bar{0}$, where H_r is H with all the elements of the first row replaced by zero [29, 21].

A group reuse is defined between multiple references to the same array. As noted by Gannon et al. [9], significant group data reuse *can* be exploited only if it occurs between *uniformly generated references* (UGR), that is, the references with the same access matrices. For instance, given two uniformly generated references of the forms $H\bar{i} + \bar{c_1}$ and $H\bar{i} + \bar{c_2}$ to the same array where $\bar{i}$ is the iteration vector, a *group-temporal reuse vector* $\bar{r}$ can be found by solving the set of simultaneous equations $H\bar{i'} + \bar{c_1} = H\bar{i''} + \bar{c_1}$ and $\bar{r} = \bar{i''} - \bar{i'}$. Similarly, a *group-spatial reuse vector* can be defined as $\bar{r} = \bar{i''} - \bar{i'}$, where $\bar{i'}$ and $\bar{i''}$ are solution to the system $H_r\bar{i'} + \bar{c_1} = H_r\bar{i''} + \bar{c_2}$ (H_r as defined above).

A collection of individual reuse vectors is referred to as a *reuse matrix*. The purpose of an optimization strategy is then to transform the loop nest such that the reuse matrix is in a desired form. The most desirable reuse vector is of the form $\bar{r} = [0, ..., 0, 1]^T$, which indicates that the data reuse in question is *carried* by the *innermost loop*. Li shows in his thesis [21] that applying a loop transformation represented by a square matrix T transforms a given reuse vector $\bar{r}$ to $T\bar{r}$. The purpose of the locality optimization (based on reuse vectors) is to find a suitable T that would transform a given reuse vector to $[0, ..., 0, 1]^T$. In those cases where a reuse matrix contains different reuse vectors, a strategy might be optimizing the most frequently occurring reuse vectors first [21].

An *iteration space* (*loop*) *transformation* can be represented by an $n \times n$ non-singular transformation matrix T, and maps the iteration $\bar{i'}$ of the original loop nest to the iteration $T\bar{i'}$ of the transformed loop nest [29, 21, 31]. On applying the transformation, an original array reference $H\bar{i} + \bar{c}$ is transformed to a new array reference $HT^{-1}\bar{j} + \bar{c}$, where $\bar{j}$ is the new iteration vector. Note that a loop transformation modifies the access matrix but *not* the offset vector. The loop transformation also affects the loop bounds of the iteration space that can be computed (after the transformation) using techniques such as Fourier-Motzkin elimination [31].

3 DRV: Data Relation Vectors

Informally, a *data relation vector* (DRV) captures the relation between two data elements accessed in the same or consecutive loop iterations. Consider the generic loop nest below written in an abstract form that uses two *uniformly intersecting references* (UIR)[1] to the same array.

```
for ī = ī_l, ī_u
  {U[Hī + c̄1], U[Hī + c̄2]}
end for
```

A *group-DRV* is defined as the vector difference between the two data elements accessed by these two references in a *fixed* value of $\bar{i}$. Therefore, for a given $\bar{i'} \in [\bar{i_l}, \bar{i_u}]$, from $(H\bar{i'} + \bar{c_1}) - (H\bar{i'} + \bar{c_2}) = \bar{c_1} - \bar{c_2}$, we can write the data relation vector $\bar{s}$ as $\bar{s} = \bar{c_1} - \bar{c_2}$. For convenience, whenever possible, we express any DRV as a lexicographically positive vector.

[1] Two references are said to be uniformly intersecting if they are uniformly generated and actually access a common data element in the given iteration space [1]. For example, references $U[2i]$ and $U[2i + 1]$ are uniformly generated but *not* uniformly intersecting.

Note that even in the uniformly intersecting references (UIR) case, DRV for this nest can be different from both data dependence vector and data reuse vector.

Since a DRV represents the relationship between points (elements) of the same data space (array), they can be manipulated using data space (array layout) transformations. This is conceptually similar to reuse or dependence vectors that can be manipulated using iteration space (loop) transformations. Consequently, *we can select a suitable data space transformation to obtain a desired DRV for a given objective.* In the following, we discuss four different optimizations based on DRVs: optimizing group-spatial reuse (Section 3.1), optimizing self-spatial reuse (Section 3.2), enhancing intra-tile and inter-tile locality in tiling (Section 3.3), and eliminating redundant communication on parallel architectures (Section 3.4).

3.1 Optimizing Group-Spatial Reuse

Consider the following example:

Example 1:

```
for i = 3, 100
  for j = 1, 200
    {U[i, i + j], U[i − 2, i + j − 1]}
  end for
end for
```

Since $\bar{c_1} = [0, 0]^T$ and $\bar{c_2} = [-2, -1]^T$, the DRV here is $\bar{s} = \bar{c_1} - \bar{c_2} = [2, 1]^T$. Note that in this example, the group-temporal data reuse vector (which is the same as the data dependence vector) can be found by solving the following system, where $\bar{r} = [r_1, r_2]^T$ is the reuse vector:

$$\exists\ (i_1, j_1),\ (i_2, j_2):\quad 3 \le i_1, i_2 \le 100,\quad 1 \le j_1, j_2 \le 200,$$
$$i_1 = i_2 - 2,\ i_1 + j_1 = i_2 + j_2 - 1,\ r_1 = i_2 - i_1,\ r_2 = j_2 - j_1$$

A solution to this system is $\bar{r} = [r_1, r_2]^T = [2, -1]^T$, which is different from the DRV ($[2, 1]^T$). It is easy to see that in this example, the DRV is a diagonal vector which is *not* very good from the layout point of view as conventional languages store multi-dimensional arrays in either row-major or column-major order. Assuming that array U is stored column-major, a more suitable DRV would be $[1, 0]^T$; that is, whenever we access an element (using the first reference), the element that will be acessed through the second reference in the *same* iteration will be the *next one* in the *same* column (and hopefully in the same cache line). In the following, we show that a desired (from an optimization point of view) DRV can be obtained using data space transformations.

Lemma 1: A data space transformation represented by a non-singular square matrix M maps a given DRV $\bar{s}$ to a new DRV $\bar{s'} = M\bar{s}$.

Proof: It is known from previous work [20] that a data transformation M applied to an array transforms the array element in $\bar{a}$ (array index) to $M\bar{a}$. Let $\bar{a_1} = H\bar{i} + \bar{c_1}$ and $\bar{a_2} = H\bar{i} + \bar{c_2}$. Suppose that, after the transformation, $\bar{a_1}$ and $\bar{a_2}$ are mapped onto $\bar{a_1}'$ and $\bar{a_2}'$, respectively. Consequently, $\bar{a_1}' = M\bar{a_1} = MH\bar{i} + M\bar{c_1}$ and $\bar{a_2}' = M\bar{a_2} = MH\bar{i} + M\bar{c_2}$. Therefore, the new DRV, $\bar{s'} = \bar{a_1}' - \bar{a_2}' = M\bar{c_1} - M\bar{c_2} = M(\bar{c_1} - \bar{c_2}) = M\vec{s}$ as M is non-singular. □

Now, we can express our problem of obtaining a new DRV $\bar{s'} = [1, 0]^T$ for Example 1 as one of finding a data transformation matrix M such that $M[2, 1]^T = [1, 0]^T$. It is easy to see that a data transformation matrix that satisfies this requirement is

$M = \begin{bmatrix} 1 & -1 \\ 1 & -2 \end{bmatrix}$. Once a data transformation matrix is determined, generating code (which involves transforming subscript expressions, modifying array declaration, and inserting necessary remapping loops in the code if necessary) is a mechanical process [20, 23]. In this example, after transforming each reference (by pre-multiplying with M), we obtain the following loop nest, where U' represents the new (transformed) array:

```
for i = 3, 100
  for j = 1, 200
    {U'[-j, -i - 2j], U'[-j - 1, -i - 2j]}
  end for
end for
```

Note that both the references access the same column (indexed using $-i-2j$) and the group-spatial locality is improved. In languages that do not allow negative array indices, we can always use shift-type of data transformations by adding a constant term to each subscript position. This small example shows that, in the case of UGRs, computing DRV is easier than computing reuse vectors (which may require solving an equational system) and the DRV is a natural abstraction that can be manipulated by data space transformations.

A given array in a loop nest may have multiple DRVs, each between a pair of references. Assuming that $\bar{s_1}$, $\bar{s_2}$, ..., $\bar{s_k}$ are the DRVs for given nest, we can build a *data relation matrix* (DRM) whose columns are individual DRVs. It might also be important to order the columns of DRM, in particular, when one wants to optimize the DRM (that is, to bring it to a specific, desired form) but not all of the DRVs in it can be optimized at the same time. Based on convenience, we order the DRVs from left to right in decreasing order of the frequency of occurrence; in other words, we use the frequency of occurrence as the priority in our approach. The DRM built from $\bar{s_1}$, $\bar{s_2}$, ..., $\bar{s_k}$ is written as $S = [\bar{s_1}; \bar{s_2}; ...; \bar{s_k}]$. It can easily be shown using Lemma 1 that, if M is a non-singular data transformation and S is the data relation matrix, then, after the transformations, the new DRM is $S' = MS$.

Assuming a *column-major* memory layout, given a DRM, the goal of locality optimization is to bring it to the following form

$$S' = \begin{bmatrix} \delta_1 & \delta_2 & \cdots & \delta_{k-1} & \delta_k \\ 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix},$$

where each δ_i is a small constant (preferably 1). If this is not possible, we want to optimize the DRVs of highest priority. Consider the following example nest.

Example 2:

```
for i = 2, 100
  for j = 2, 99
    {U[i - 1, j], U[i, j - 1], U[i + 1, j - 2]}
  end for
end for
```

The data relation matrix for array U can be written as $S = \begin{bmatrix} 1 & 2 \\ -1 & -2 \end{bmatrix}$. Note that although there are three DRVs, since the DRV $[1, -1]^T$ occurs twice, it is placed into the leftmost position (Consequently, we have only two distinct DRVs). An alternative way of building a DRM is to use (as columns of the DRM) only the *basis vectors* of the set consisting of all DRVs. In that case, since $[1, -1]^T$ and $[2, -2]^T$ are linearly dependent, the DRM can be represented using only the basis vector, which is $[1, -1]^T$. Our current implementation, however, uses the former definition for simplicity. Note that using the data transformation matrix $M = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$, the new DRM can be found as $S' = \begin{bmatrix} 1 & 2 \\ 0 & 0 \end{bmatrix}$. As before, we denote the layout-transformed version of array U as U'. The transformed references then (from left to right) are $U'[2i+j-2, i+j-1]$, $U'[2i+j-1, i+j-1]$, and $U'[2i+j+1, i+j-1]$. It should be noted that while in the original nest (in a given fixed iteration) all three references access three different columns (leading to poor group-spatial locality), in the optimized nest, for a given iteration, all three references access the same column (indexed using $i+j-1$). Moreover, the elements accessed by these references are very close to each other, mapping (hopefully) onto the same cache line.

Note that if the original layout of U was row-major (instead of column-major) in this example, the desired DRV would be $[0, 1]^T$ (instead of $[1, 0]^T$) and the layout-transformed array references would be $U'[i+j-1, 2i+j-2]$, $U'[i+j-1, 2i+j-1]$, and $U'[i+j-1, 2i+j]$. Note that these three transformed references access the same row (indexed using $i+j-1$ for a given fixed iteration vector $[i, j]^T$), which means good spatial locality for a row-major memory layout. Contrast this with the original case (assuming row-major layout) where three references access three different rows indexed using $i-1$, i, and $i+1$.

It is easy to see that the DRV concept can also be used for optimizing *non-uniformly intersecting references* (non-UIRs) as well. As an example, let us consider the following code.

Example 3:

```
for i = 1, 100
  for j = 2, 99
    {U[2i, 2j], U[2i - 1, j - 2]}
  end for
end for
```

The two references to array U do *not* have the same access matrix (i.e., they are not uniformly generated). The DRV in this example can be represented as $[1, j+2]^T$, where $2 \leq j \leq 99$. Using a data transformation matrix of the form $M = \begin{bmatrix} -2 & 1 \\ 1 & 0 \end{bmatrix}$, the new DRV is $[j, 1]^T$, where $2 \leq j \leq 99$. Note that, assuming a column-major memory layout, this new DRV is better than $[1, j+2]^T$ as it causes the two references access (in the transformed code) the neighboring columns (which may be useful in optimizing page locality or cache locality when the array size is not very large. It should be noted that a DRV such as $[j, 1]^T$ indicates that two references access consecutive columns. This is because the second entry in the DRV is 1. However, depending on the current value of the loop index variable j, these references can access rows far apart from each other. Contrast this with a *constant* DRV such as $[1, 0]^T$ which indicates that the references access the same column (the second entry is zero) but consecutive rows (the first entry is one). In a column-major layout, the elements accessed by these two references are consecutive.

3.2 Optimizing Self-Spatial Reuse

In this subsection, we illustrate how the DRV abstraction can be used for improving self-spatial data reuse. Consider again the generic loop nest given at the beginning of Section 3. Let us focus on only the first array reference. Using the said reference, two *consecutive* loop iteration vectors that do *not* cross loop boundaries (differ by a unit vector) $\bar{i'}$ and $\bar{i'}_+$, access the data elements

$\bar{a_1} = H\bar{i'} + \bar{c_1}$ and $\bar{a_1}_+ = H\bar{i'}_+ + \bar{c_1}$. So, a *self-DRV* can be defined as the difference between $\bar{a_1}_+$ and $\bar{a_1}$.

Lemma 2: For a given array reference represented by H and $\bar{c_1}$, the self-DRV is the last column of H.

Proof: Let $\bar{a_1} = H\bar{i'} + \bar{c_1}$ and $\bar{a_1}_+ = H\bar{i'}_+ + \bar{c_1}$ as above. Then, the self-DRV $\bar{s} = (H\bar{i'}_+ + \bar{c_1}) - (H\bar{i'} + \bar{c_1}) = H(\bar{i'}_+ - \bar{i'})$. Since $\bar{i'}_+ - \bar{i'} = [0, 0, ..., 0, 1]^T$, $\bar{s} = H[0, 0, ..., 0, 1]^T = \bar{h}_l$, where $\bar{h}_l$ is the last column of H. □

Similar to group-DRVs, self-DRVs can also be re-shaped using data space transformations. Consider now the first array reference in Example 2 assuming a column-major memory layout. Since the self-DRV $\bar{s}$ is $[0, 1]^T$ (the last row of H), we can use a data transformation vector such as $M = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, and obtain the new self-DRV as $\bar{s'} = [1, 0]^T$. Note that this transformation corresponds to converting the memory layout of the array from column-major to row-major [20].

It should be noted that obtaining a desired self-DRV (that is, optimizing self-spatial reuse) can sometimes conflict with obtaining a desired group-DRV (that is, optimizing for group-spatial reuse). For instance, in Example 2 above, the necessary data transformation matrices for desired group-DRV and desired self-DRV are different from each other. Note that optimizing for first self-spatial locality and then group-spatial locality may not work in general. Continuing with Example 2, assuming that the memory layout of array U is column-major, both group-spatial locality and self-spatial locality are not good (i.e., three references access three different columns and each reference—when considered alone—accesses different columns in successive iterations of the inner loop j). While one might think of first optimizing self-spatial reuse using a transformation matrix such as $M' = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and then optimizing group-spatial reuse with $M'' = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$, this second transformation would destroy the good self-spatial locality obtained by the first transformation. In general, this problem can be handled by giving priority to each DRV (self or group) and by ordering all the DRVs (according to their priorities) when building the DRM.

3.3 Tiling

Loop tiling (also called *blocking*) [31, 29, 27, 11] is a widely-used compiler technique to improve data locality and/or parallelism. While it can be used to exploit data reuse for any level in a given memory hierarchy, we focus here on tiling for optimizing cache performance. The idea is that data structures that are too big to fit in the data cache are broken up into smaller pieces (called *blocks* or *tiles*) that will fit in the cache. In other words, instead of operating on individual elements of arrays, tiling performs computation on blocks. Unfortunately, one of the problems with many tiling algorithms is that they do *not* take into account the relationship between data elements accessed by iteration space tiles. The main concern instead is selecting a suitable tile size so that the data accessed (possibly from different arrays) by a given iteration space tile should fit in the cache and should not cause many conflict misses during the execution of the said iteration space tile. In addition, most work in choosing a good tile shape is control-centric, i.e., based on legality constraints imposed by data dependence vectors.

However, as will be shown in subsequent discussion, it is important to take the shape of the data portion accessed by a given iteration tile into account. In this subsection, we use the term *data tiles* for the portions of arrays accessed by a given iteration space tile. As an example, let us consider the loop nest shown below.

Example 4:

```
for i = 2, N
  for j = 2, N
    {U[i-1, j-1], U[i, j]}
  end for
end for
```

Figure 1(a) shows an iteration space tile for this nest (assuming that no data dependence other than possibly input dependence exists). The right part of Figure 1(a), on the other hand, shows the data space and the array elements accessed by the 3×3 tile shown in the bottom of the left part of Figure 1(a). There are 18 references made from within this tile (two references per iteration point in the tile) and these references access total 14 distinct array elements. Now, consider the alternative iteration space tile shown in Figure 1(b). Twelve distinct array elements accessed by this tile are shown on the right part of the same figure. It is easy to see that the tile given in Figure 1(b) causes more *intra-tile reuse* than the one in Figure 1(a).

In the following, we explain how DRVs can be used to select *tile shapes for high intra-tile data reuse.* Informally, the idea is to select an iteration space tile shape that captures as many DRVs as possible inside the corresponding data tile (i.e., minimize the number of DRVs crossing data tile boundaries). This data-oriented strategy also makes use of a polyhedral tool (the Omega library from University of Maryland [16]). Our approach consists of three steps:

(1) Determining the data relation vectors on the data space,

(2) Selecting a suitable data tile shape using the DRVs found in the previous step, and

(3) Selecting a suitable iteration tile considering the data tile shape found in the second step.

The first step is straightforward and the left part of Figure 1(c) shows the only DRV (which is $[1, 1]^T$) on the data space of this example. Our ultimate objective is to select an iteration tile shape such that the total number of array elements within the corresponding data tile is minimum. Since a data relation vector represents the relation (or distance) between data points, the said objective can be accomplished if the fewest number of DRVs is shared between two different (neighboring) data tiles. In other words, the number of out-going (resp. incoming) data relation vectors from (resp. to) a data tile should be minimized (Contrast the two different data tile shapes shown on the right part of Figure 1(c)). This is achieved in the second step above. Once we select a suitable data tile, we need to find the corresponding iteration space tile and transform (tile) the code accordingly. We assume for now that the loop nest to be optimized does *not* contain any data dependence.

Each *boundary* of a data tile can be defined by a vector *perpendicular* to it. In an m-dimensional *data space,* any m linearly independent vectors $\bar{l_1}, \bar{l_2}, ..., \bar{l_m}$ (where $\bar{l_j}$ is the vector perpendicular to the jth boundary) define a *data tile.* If $P = [\bar{p_1}; \bar{p_2}; ...; \bar{p_m}]$ is a matrix where each column is a tile boundary, then $L = P^{-1}$, where L is a matrix whose rows are $\bar{l_1}, \bar{l_2}, ..., \bar{l_m}$. Further, if we let S be the data relation matrix (DRM), a nonzero entry s'_{ij} in $S' = LS$ indicates that the jth DRV pokes the ith tile boundary. The problem then can be formulated as one of determining an L matrix (or equivalently a P matrix) such that the sum of the absolute values of the entries in S' is minimum.

Continuing with our current example, we have $S = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and $L = \begin{bmatrix} l_{11} & l_{12} \\ l_{21} & l_{22} \end{bmatrix}$. Consequently, $S' = LS = \begin{bmatrix} l_{11} + l_{12} \\ l_{21} + l_{22} \end{bmatrix}$. Thus, we want to minimize $|l_{11} + l_{12}| + |l_{21} + l_{22}|$. Also, the L

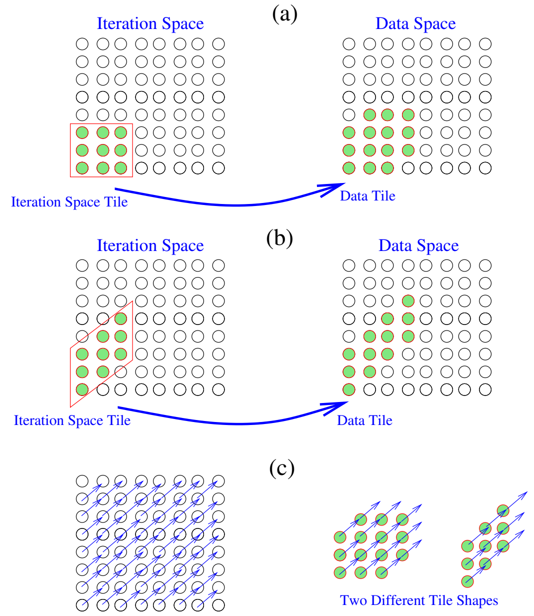

Figure 1: (a-b) Two different iteration space and data space tiles for Example 4. (c) DRVs and two different data tiles.

matrix should be invertible (i.e., non-singular). A solution in this case is $l_{11} = 1, l_{12} = 0, l_{21} = -1$, and $l_{22} = 1$, which gives $L = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \Rightarrow P = L^{-1} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. Note that the data tile implied by this P is the one depicted in Figure 1(b). Note also that in this case $|l_{11} + l_{12}| + |l_{21} + l_{22}| = 1$.

While it might be difficult to determine an L matrix automatically (as this process involves a *non-linear* constraint of non-zero determinant), we can reduce the search space by fixing the shape of L at a specific form (with non-zero determinant). One alternative is a lower-triangular *integer* matrix with all diagonal entries of 1 (that is, a specific form of *unimodular matrix* [31, 27]).

For our current example with the two-dimensional array U, the problem of finding an L matrix with maximum intra-tile reuse can be expressed as that of finding a matrix L of the form $L = \begin{bmatrix} 1 & 0 \\ a & 1 \end{bmatrix}$ such that $S' = LS = \begin{bmatrix} 1 \\ a+1 \end{bmatrix}$ has entries whose sum (of absolute values) is minimum. Consequently, the problem is selecting an a to minimize $|a + 1| + 1$. It is easy to see that $a = -1$ is the value that minimizes that expression. Note that the fact that we focus only on integer entries and that the determinant of L is 1 guarantees that both L and P are integers. In this example, using $a = -1$ results in an integer $P = L^{-1}$ matrix, which gives us the same L matrix determined as above.

After selecting a data tile, the next step is determining the corresponding iteration space tile. This can be done using a technique similar to the one proposed by Kodukula et al. [18]. Due to lack of space, we provide only a brief discussion here. The main idea here is to select a data tile and to execute all the iterations that use the elements in the current data tile. While Kodukula et al. [18] fix their data tile shapes to be rectangular, we show here that, for best intra-tile reuse, the data tile shape should be tailored to the access pattern. Going from data tile to iteration space tile can be done using a polyhedral tool. That is, to determine the iterations in the iteration space tile that corresponds to the data tile $V = \{\bar{v}_1, \bar{v}_2, ..., \bar{v}_q\}$ of q elements (where each $\bar{v}_i$ is an array index in the data tile), we enumerate the set $I = \{\bar{i} | H\bar{i} + \bar{c} \in V$ and $\bar{i_l} \leq \bar{i} \leq \bar{i_u}\}$ and execute those iterations ($\bar{i_l}$ and $\bar{i_u}$ are the bounds for the iteration space). Our current implementation uses the Omega library for this enumeration.

Since our goal is to derive the shape of iteration space tiles from data tiles, it is important to check the legality of the iteration space tiles when data dependences exist. Currently, we generate a set of candidate data space tiles, from which we select only the legal tiles. We are exploring the option of generating a succinct representation of valid tile shapes and using that in conjunction with data space tile shapes. It is possible to use data tile shapes for each array in the code in this framework. However, we currently employ the following heuristic. For the most important array (from locality point of view), we determine the data tile shape; this induces legal iteration space tile shapes for the loop nest in question. The iteration tile shapes derived this way in turn impose data tile shapes for other arrays in the nest.

3.4 Minimizing Communication

In a message-passing machine, the communication cost incurred under an iteration space partitioning (i.e., mapping iteration points into processors) depends among other things on the number of data transfers between processors. Even in fully-parallel computations, one has to be very careful in partitioning the data space. Consider the following nested loop.

Example 5:

```
for i = 2, 100
  for j = 2, 100
    U[3i, 2j] = U[3i − 2, 2j − 1];
  end for
end for
```

This loop is fully parallel, but standard row, column, or block partitions of the data space (array U) between processors would entail unnecessary communications. Instead, a communication-free partitioning should be along the direction given by the vector $[2, 1]^T$, the *data relation vector* for the references in this loop nest. Of course, using this vector and an appropriate data transformation, the array in this nest can be transformed to another array where the partitions are by columns, rows, or blocks. In cases where there are more than one DRVs in the nest, selecting a data space decomposition that minimizes communication involves considering all DRVs. For example, if, in a given nest, there are three data relation vectors, $[0, 2]^T$, $[3, 0]^T$, and $[3, 2]^T$, to minimize the inter-processor communication, we need to choose a block partitioning along the direction given by the vector $[3, 2]^T$. Note that the process explained above for minimizing inter-tile DRVs (for maximizing locality) can also be used here. Ramanujam and Sadayappan [26] show how a similar data-oriented abstraction can be used for minimizing communication on message-passing distributed-memory machines.

4 Unified Optimizations

In this section, we present a unified locality optimization technique that employs *both* data relation vectors and reuse vectors. The idea behind the technique is that optimizing data locality involves selecting a suitable execution order for loop iterations and choosing appropriate memory layouts for multi-dimensional data structures. Since, loop transformations manipulate the execution order of loop iterations and consequently reuse vectors, and data transformations manipulate the memory layouts and data relation vectors, reuse vectors and data relation vectors are complementary abstractions for representing and optimizing locality.

As noted by O'Boyle and Knijnenburg [23], the most important difference between loop and data transformations is the scope of application. Loop transformations can be applied to one loop at a time; in other words, each loop nest can be optimized (using loop transformations) independently from other nests. Data transformations, on the other hand, operate on arrays. Since a given array can be shared (accessed) by a number of loop nests, determining best memory layout for a given array requires considering the access patterns of *all* the loops to the said array. This global view is necessary as different loop nests may demand different memory layouts (and, of course, different data transformations) for a given array. Motivated by this observation, our unified approach to optimizing locality works as follows.

(1) We first determine for each array all the DRVs considering all the nests accessing the array. This gives us a matrix called *global data relation matrix* (GDRM) for each array. Then,

(2) Using its GDRM, for each multi-dimensional array, we determine a suitable memory layout and corresponding data transformation that implements it, and

(3) Next, for each nest, we determine a suitable loop transformation taking into account the memory layouts determined in the previous step.

Note that this approach first fixes data transformations and then determines loop transformations. This order is appropriate as data transformations for arrays affect multiple loop nests, so they need to be considered first. In the following, we show that, after a GDRM is built for each array, steps 2 and 3 can optimize group-spatial and self-spatial reuses, respectively. Let us consider the following example program fragment that contains two loop nests and accesses three different arrays.

Example 6:

```
for i = 3, 100
  for j = 3, 100
    {U[i,j], U[i,j-1], W[i,i+j], W[i-2,i+j]}
  end for
end for
for i = 2, 100
  for j = 2, 100
    {U[i+j+1,j+1], U[i+j+1,j], W[i,i+j]}
        {W[i-1,i+j-2], V[j,i], V[j-1,i+1]}
  end for
end for
```

Considering both the nests and only *group* data relation vectors, the GDRMs are $S_u = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $S_v = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, $S_w = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}$, for arrays U, V, and W, respectively. Note that the entry in S_u occurs in both the nests whereas the entry in S_v comes from the second nest only. Note also that S_w has two conflicting entries, each coming from one nest. Now, using these GDRMs and assuming column-major memory layout as default, we can select suitable data transformation matrices as follows: $M_u = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $M_v = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, and $M_w = \begin{bmatrix} 1 & 0 \\ 2 & -1 \end{bmatrix}$. Note that while optimizing layouts we resolved the conflict in determining layout of W in favor of the second loop nest. In general, it is possible to use a sophisticated conflict resolution scheme (e.g., [13]). After applying the data transformations represented by the matrices given above, we obtain the following fragment.

```
for i = 3, 100
  for j = 3, 100
    {U'[j,i], U'[j-1,i], W'[i,i-j], W'[i-2,i-j-4]}
  end for
end for
for i = 2, 100
  for j = 2, 100
    {U'[j+1,i+j+1], U'[j,i+j+1], W'[i,i-j]}
        {W'[i-1,i-j], V'[j,i+j], V'[j-1,i+j]}
  end for
end for
```

In the second loop nest in this code, a given iteration vector $[i, j]^T$ accesses two consecutive array elements from each array, therefore, as far as group-spatial locality is concerned, this nest is fully optimized. As for the first nest, only the accesses to the original array U are optimized (as a result of our conflict resolution scheme). In contrast, self-spatial locality for the references shown is not as good. In the final step of the optimization technique sketched above, we use a loop transformation for each nest to optimize self-spatial and temporal reuse for the arrays it accesses. For instance, if we focus on references $W'[i, i-j]$ and $W'[i-1, i-j]$ in the second nest, while their group-spatial locality is good (as the access two consecutive elements from the same column — indexed by $i - j$ — for a given iteration vector), individually they access the array row-wise, leading to poor self-spatial reuse. However, if we transform this nest using a loop transformation represented by the non-singular matrix $T = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$, these two references become $W'[i' + j', i']$ and $W'[i' + j' - 1, i']$, respectively. Note that both the references exhibit good self-spatial locality after this loop transformation. Of course, in general, when selecting a loop transformation, all the arrays accessed in the nest should be considered.

The main idea is, therefore, to use first data transformations for optimizing group-spatial reuse, and then loop transformations for optimizing self-spatial/temporal reuse. In our current example, after applying data transformations, we have the following *reuse vectors* for the first nested loop: $\bar{r_u} = [0, 1]^T$ and $\bar{r_w} = [1, 1]^T$. Since the reuses are in orthogonal directions, it is not possible to optimize self-spatial locality for both the arrays. We can leave this nest as it is, thereby allowing only array U' to have self-spatial locality. In the second nested-loop, on the other hand, we have the following spatial reuse vectors: $\bar{r_u} = [1, -1]^T$, $\bar{r_v} = [1, -1]^T$, and $\bar{r_w} = [1, 1]^T$. Since $[1, -1]^T$ occurs twice, a *dependence-sensitive* locality optimization framework that uses only loop transformations such as [21] would attempt to optimize for this vector by selecting a loop transformation matrix such as $T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. The final fragment is as follows.

```
for i = 3, 100
  for j = 3, 100
    {U'[j,i], U'[j-1,i], W'[i,i-j], W'[i-2,i-j-4]}
  end for
end for
for i' = 4, 200
  for j' = max(i'-100,2), min(i'-2,100)
    {U'[i'-j'+1,i'+1], U'[i'-j',i'+1], W'[j',2j'-i']}
      {W'[j'-1,2j'-i'], V'[i'-j',i'], V'[i-j'-1,i']}
  end for
end for
```

Note that in the first nest we have self-spatial locality and group-spatial locality for U'. Further, we have group-spatial locality for all arrays and self-spatial locality for U' and V' in the second nest. It should also be noted that loop optimization for improving self-spatial reuse does *not* distort the good group-spatial locality obtained by a previous data transformation, because we assume uniformly intersecting references and it is known that a loop transformation does *not* change the offset vector part of a given reference. It should be noted that, in contrast to reuse vectors, global data relation vectors capture the access patterns of multiple loop nests to a given array and allow global (procedure-wide) optimizations. The unified optimization technique discussed above is just one alternative and we expect that more powerful optimization techniques can be developed using a suitable mix of reuse and relation vectors.

5 Experimental Evaluation

We present performance results on a single processor of the SGI Origin 2000 using the applications given in Table 1(a). Each processor of the Origin, an R10000, is a 195 MHz four-way superscalar RISC processor. It can fetch and decode four instructions per cycle to be run on its five independent, pipelined execution units: a nonblocking load store unit, two 64-bit integer ALUs, a 32-/64-bit pipelined floating-point adder, and a 32-/64-bit pipelined floating-point multiplier. Our experimental suite contains three image processing programs (`full_search`, `matvec`, and `phods`) and nine regular scientific codes, all manipulating large multi-dimensional arrays. On the Origin, an L1 (32 KB) cache hit takes 2 or 3 cycles and an L1 cache miss satisfied by L2 (4 MB) cache hit takes 8 to 10 cycles. On the other hand, an L2 cache miss satisfied from main memory (without TLB miss) takes 75 to 250 cycles, therefore, optimizing cache locality is of critical importance.

We start by evaluating the performance of our unified strategy where we first use data transformations to optimize group-spatial reuse, and then use loop transformations to optimize self-spatial and temporal reuse. Throughout this section, we refer to this version as `unified`. We also compare our approach to two other locality optimization schemes: one that uses *only* loop transformations for optimizing both temporal and spatial locality (called `loop`) and one that uses *both* loop and data transformations for optimizing only self (spatial and temporal) reuse (called `data`). The first approach is very similar to that proposed in [21] and the second one is the technique proposed in [13]. By comparing our technique to `loop`, we investigate the benefits due to employing data transformations and by comparing our technique to `data`, we are able to evaluate the benefits due to taking group reuse into account. We then evaluate, using an example code, the tiling strategy explained in Section 3.3.

5.1 Compilation Times

All these three optimization techniques are implemented using SUIF 1.x from Stanford University [2] and the resulting codes (generated by each version) are compiled using the native compiler on the Origin. The compiler flag `O3` is used, but loop permutations and loop tiling are explicitly disabled (using compiler flags) to isolate the influences of the three techniques (However, other loop transformations, prefetching, and all low-level optimizations are enabled).

Before presenting performance numbers, we give in Figure 1(b) the compilation times *in milliseconds* for all three techniques. The time spent within the commercial compiler is *not* included in these entries, but it is quite small and almost invariant across different codes and different versions. We observe from these results that our approach reduces the compilation time by 7.6% and 22.3%, respectively, over `loop` and `data`. This is because `loop` uses reuse vectors for both self and group reuse, and `data` uses two separate abstractions, both costlier than data relation vectors. Our unified approach, on the other hand, uses data relation vectors for handling group spatial reuse and uses reuse vectors only for self spatial reuse. Also, as compared to the `loop` version, our approach applies loop transformations much less frequently as most of the locality improvement comes from global layout optimization. Although the reduction on compilation time over `loop` is not too much, it should be noted that our approach uses both loop and data transformations and generates better code than `loop` as will be shown below.

5.2 Performance

We compare performance of the original (unoptimized) codes and optimized codes (`loop`, `data`, and `unified`) using five different metrics: L1 Cache Line Reuse, L2 Cache Line Reuse, L1 Cache Hit Rate, L2 Cache Hit Rate, and MFLOPS, all of which are obtained using performance counters on the R10000 processor. L1 Cache Line Reuse is the number of times, on average, that a primary data cache line is used after it has been moved into the cache. It is calculated as graduated loads plus graduated stores minus primary data cache misses, divided by primary data cache misses. L2 Cache Line Reuse is the number of times, on average, that a secondary data cache line is used after it has been moved into the cache. It is calculated as primary data cache misses minus secondary data cache misses, divided by secondary data cache misses. L1 Cache Hit Rate is the fraction of data accesses that are satisfied from a cache line already resident in the primary data cache. It is calculated as 1.0 - (primary data cache misses divided by the sum of graduated loads and graduated stores). L2 Cache Hit Rate is the fraction of data accesses that are satisfied from a cache line already resident in the secondary data cache. It is calculated as 1.0 - (secondary data cache misses divided by primary data cache misses). MFLOPS is the ratio of the graduated floating point instructions and the total program run time. It should be noted that although a multiply-add carries out two floating point operations, it only counts as one instruction, so this statistic may underestimate the number of floating point operations per second.

Figure 2 shows the L1 Cache Line Reuses for different versions (`unoptimized` is the original code without any linear loop and data optimization). It is easy to see that there is not much difference between different versions as far as the L1 cache performance is concerned. For instance, our approach is, on the average, only 5% better than `data`. As shown in Figure 3, L1 Cache Hit Rates also follow the same trend. However, when we consider the L2 performance (Figures 4 and 5), this picture changes. Our approach results in 30% and 62% better L2 Cache Line Reuse than `data` and `loop`, respectively. Thus, L2 cache performance is the main contributer to the difference in the runtime performances of different versions.

Figure 6 shows the MFLOPS rates for different versions. Our approach improves upon `loop` and `data` by approximately 48% and 26%, respectively, on the average. In `eflux`, non of the optimizations are successful, and in `adi`, `mxm`, and `cholesky`, our approach generates the same output code as `data`. The main reason for that is the lack of opportunities for exploiting group spatial reuse exclusively. On the other hand, in the image processing codes (`full_search`, `matvec`, and `phods`), our approach takes advantage of group spatial reuse. The additional benefit of our technique in codes such as `amhmtm`, `bmcm`, and `btrix` is due to the fact that our approach considers all nests simultaneously before deciding any memory layouts whereas `data` optimizes one nest at a time and propagates layouts across nests.

So far, we have excluded tiling from all the performance results presented. Figure 7 gives the MFLOPS rates for four different versions of `phods` with ten different input sizes. The first two versions are `unoptimized` and `unified` which have already been explained above. `unified+` is the `unified` followed by traditional loop tiling. `unified++` is, on the other hand, the `unified` version followed by the tiling strategy discussed in Section 3.3. In these last two versions, each loop is tiled with a size of 20. The performance of the unoptimized code (averaged over ten input sizes)

Table 1: Benchmark codes and their compilation times.

(a): Brief description of benchmarks used

Benchmark	Source	Data Size (MB)	Brief Description
adi	Livermore	48.0	Alternate Direction Integral
amhmtm	Perfect Club	61.8	Molecular Dynamics of Water
bmcm	Perfect Club	39.9	Molecular Dynamics of Water
btrix	Spec'92	47.7	Nasa Ames Fortran Kernel
dtdtz	Perfect Club	61.8	Transonic Flow Analysis
eflux	Perfect Club	33.6	Mesh Computation
full_search	[4]	33.0	Motion Estimation Kernel
matvec	[3]	16.0	Matrix-Vector Arithmetic
mxm	Spec'92	48.0	Nasa Ames Fortran Kernel
phods	[7]	33.1	Parallel Motion Estimation
cholesky	Spec'92	56.0	Nasa Ames Fortran Kernel
vpenta	Spec'92	44.0	Nasa Ames Fortran Kernel

(b): Compilation time (in ms.)

Benchmark	loop	data	unified
adi	1,730	2,033	1,521
amhmtm	1,917	2,411	1,782
bmcm	1,910	2,305	1,805
btrix	1,915	2,250	1,613
dtdtz	2,004	2,408	2,000
eflux	1,983	2,482	1,927
full_search	1,802	1,817	1,734
matvec	1,298	1,724	1,018
mxm	1,177	1,682	1,105
phods	1,815	2,006	1,800
cholesky	1,724	2,014	1,609
vpenta	1,980	2,151	1,727

is very poor and around 24 MFLOPS only. The average rates for `unified`, `unified+`, and `unified++` are roughly 264, 273, and 286 MFLOPS, respectively. Thus, a tiling strategy that minimizes the number of data relation vectors shared by neighboring data tiles brings over 8% improvement over `unified`, which is already highly optimized.

6 Related Work

Optimizing locality of data accesses has been the main focus of several previous studies (e.g., see [31] and the references therein). Most of these techniques use linear loop transformations based on reuse vector space [21, 29] and cost-based [22, 30, 17] abstractions. The main limitation of these techniques is inherent data dependences in the code and imperfectly-nested loop structures. Some of the proposed loop-based techniques also include iteration space tiling [29, 30]. Traditional tiling techniques have difficulty in handling imperfectly-nested loops. To address this problem, Kodukula et al. [18] present a data-centric tiling strategy which is based on reasoning about the flow of data through the memory hierarchy. This technique has later been extended by Pugh and Rosser [25]. Our work differs from these studies in at least two aspects. First, we propose an abstraction that exclusively works on data space whereas the majority of the previous work use abstractions defined on iteration space. Second, our abstraction has wider applicability as it can be used for optimizing cache locality, improving the performance of tiling, or minimizing communication in message-passing architectures. Use of our approach in tiling has similar objectives as [18, 19]. However, instead of working with square and rectangular tiles, data relation vectors enable us to customize data tiles to the application's data access pattern, thereby promoting intra-tile reuse. Careful selection of tile traversals also leads to better inter-tile locality.

Several recent studies have focussed on data space transformations [20, 14, 23, 28] and on combining loop and data transformations in a unified framework [24, 8, 15, 12, 15]. Cierniak and Li [8], in particular, use an abstraction that exploits the offset difference between successively accessed elements by a given reference. They focus on self reuses and the abstraction that they use allows them to handle only column-major and row-major layouts and higher dimensional equivalents of them. Data relation vectors, on the other hand, can represent complex relations between data elements (e.g., the relations that can lead to non-traditional layouts) and can be used for optimizing self and group reuses. As shown in this paper, it is also quite easy to combine the DRVs and reuse vectors in a unified framework. Compared to studies in [13] and [12], our approach is much faster and more successful as partly shown in the experiments. Using DRV, we can represent the temporal relations between data elements (of a given array) and use this information to select globally optimal memory layouts for multi-dimensional arrays. The approaches in [24, 13, 12], on the other hand, generally optimize for a single nest at a time and propagate layouts across loop nests.

Use of DRVs for optimizing data accesses on message-passing architectures is very similar to the approach proposed by Ramanujam and Sadayappan [26]. In addition, early work on communication generation for distributed memory machines [10] and on techniques for optimizing stencil based codes for data-parallel languages [5, 6] addresses a subset of the same concerns addressed in this paper.

7 Conclusions

Exploiting data locality has played a key role in improving the performance of applications on modern architectures. Several compiler optimizations such as linear loop transformations, data layout optimizations, combination of linear loop and data layout optimizations, and tiling have been proposed for improving the locality of applications on processors. Many of these techniques use abstractions that are defined on the iteration space.

In this paper, we proposed a new abstraction called *data relation vectors* (DRVs) to capture and exploit the access relation between data elements. Using several examples, we showed how DRVs can be used for optimizing group-spatial and self-spatial reuses, for enhancing intra-tile and inter-tile locality in tiling, and for eliminating redundant communication on parallel architectures. In addition, we showed how to combine DRVs with reuse vectors in order to derive better results. Experimental results on several array-based benchmarks show that the solution that exploits DRVs outperforms other techniques. We plan to explore heuristics to reduce the size of the search space and study the effects of using a reduced representation of the data relation matrices.

Acknowledgments

The work of J. Ramanujam is supported in part by an NSF Young Investigator Award CCR–9457768.

References

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared

memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems,* 6(9):943–962, September 1995.

[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 Conference on Parallel Processing for Scientific Computing,* February, 1995.

[3] R. Bhargava, L. K. John, B. L. Evans, and R. Radhakrishnan. Evaluating MMX technology using DSP and multimedia applications. In Proc. *the IEEE Symposium on Microarchitecture,* Dallas, Texas, pp.37–46, Dec.1998.

[4] V. Bhaskaran and K. Konstantinides. *Image and Video Compression Standards – Algorithms and Architectures,* Kluwer Academic Publishers (Boston), 1996.

[5] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic optimization of communication in out-of-core stencil codes. Technical Report, Scalable I/O Initiative, Center of Advanced Computing Research, CALTECH, November 1995. A short version appeared in Proc. *ACM International Conference on Supercomputing,* Philadelphia, PA, 1996.

[6] M. Bromley, S. Heller, T. McNerney, and G. Steele Jr. Fortran at ten gigaflops: the connection machine convolution compiler. In Proc. *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 145–156, June 1991.

[7] L-G. Chen, W-T. Chen, Y-S. Jehng, and C-T. Church. An efficient parallel motion estimation algorithm for digital image processing. *IEEE Trans. Circuits and Systems for Video Tech,* 1(4):378–385, December 1991.

[8] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In Proc. *SIGPLAN Conf. Prog. Lang. Des. Impl.,* pp. 205–217, 1995.

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

[10] M. H. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.

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

[12] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In Proc. *1998 Intl. Conf. Parallel Arch. & Comp. Tech.,*, 1998.

[13] 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.

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

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

[16] 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, MD, March 1995.

[17] K. Kennedy and K. McKinley. Optimizing for parallelism and data locality. In Proc. *International Conference on Supercomputing,* ACM, New York, 1992.

[18] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In Proc. *SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.

[19] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. Imperfectly nested loop transformations for memory hierarchy management. In Proc. *International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[20] 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.

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

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

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

[24] M. O'Boyle and P. Knijnenburg. Integrating loop and data transformations for global optimisation. In Proc. *International Conference on Parallel Architectures and Compilation Techniques*, October 1998, Paris, France.

[25] W. Pugh and E. Rosser. Iteration space slicing for locality. In Proc. *International Workshop on Languages and Compilers for Parallel Computing,* August 1999.

[26] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In Proc. *the Supercomputing'89,* Reno, NV, November 1989, pages 637–646.

[27] J. Ramanujam and P. Sadayappan. Tiling multi-dimensional iteration spaces for multi-computers. *Journal of Parallel and Distributed Computing,* 16(2), pages 108–120, October 1992.

[28] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In Proc. *Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

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

[30] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In Proc. *International Symposium on Microarchitecture,* pages 274–286, Paris, France, December 1996.

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

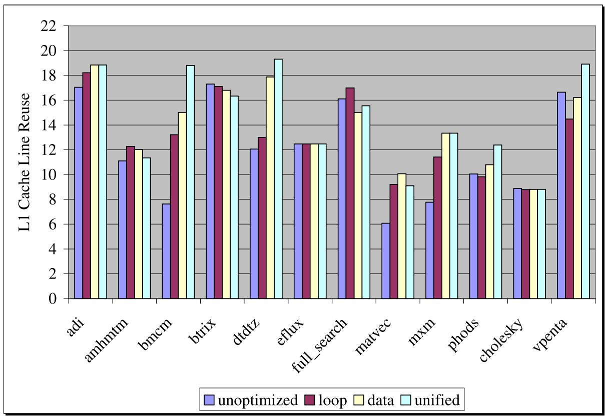

Figure 2: L1 Cache Line Reuse.

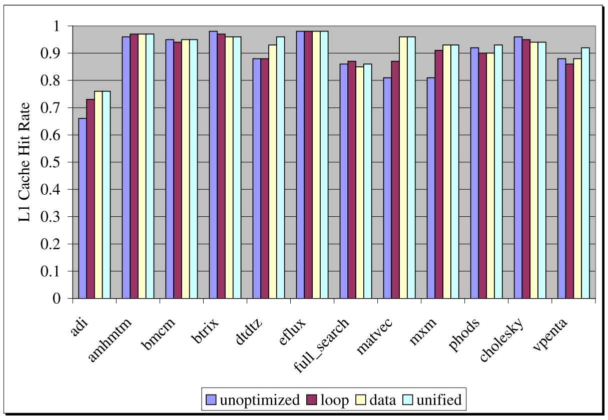

Figure 3: L1 Cache Hit Rate.

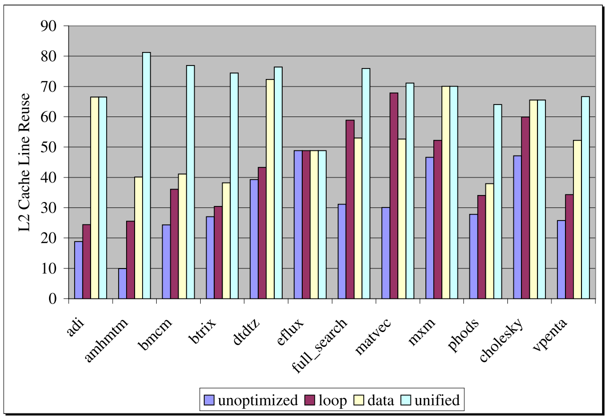

Figure 4: L2 Cache Line Reuse.

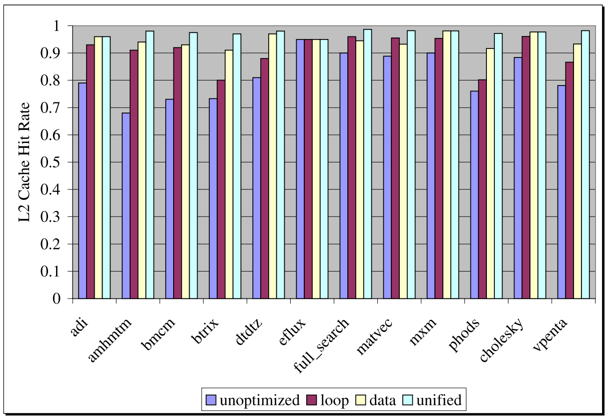

Figure 5: L2 Cache Hit Rate.

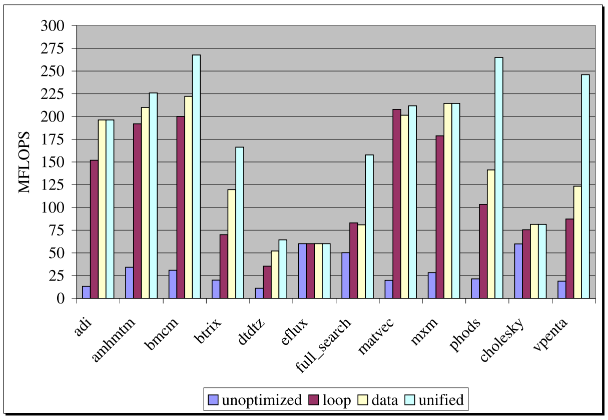

Figure 6: MFLOPS rates for different versions.

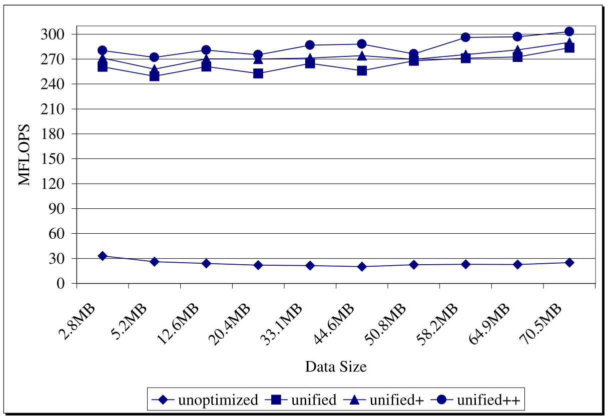

Figure 7: MFLOPS rates for different input sizes for the `phods` benchmark.