Non-Singular Data Transformations: Definition, Validity and Applications

M.F.P. O'Boyle
Department of Computer Science
The University of Edinburgh
Edinburgh EH9 3JZ
United Kingdom
mob@dcs.ed.ac.uk

P.M.W. Knijnenburg
Department of Computer Science
Leiden University
Niels Bohrweg 1, 2333 CA Leiden
the Netherlands
peterk@cs.leidenuniv.nl

Abstract

This paper describes a unifying framework for non-singular data transformations. It shows that a wide class of existing transformations may be expressed in this framework, allowing compound transformations to be performed in one step. Validity conditions for such transformations are developed as is the form of the transformed program and data. Constructive algorithms to generate data transformations for different applications are described and applied to example programs. It is shown that they can have a significant impact on program performance and may be used in situations where traditional loop transformations are inappropriate.

1 Introduction

Recent years have seen a great improvement in loop transformation theory. By using an affine representation of loops, several loop transformations have been incorporated into one single framework [18]. In [2], Banerjee shows that loop interchange, reversal and skewing can be described as unimodular transformations of the iteration space. In [14], Li and Pingali extend the class of transformations to non-singular matrices, thereby incorporating loop scaling. Furthermore, this framework has been extended to statement-wise transformations [11] and integrated with loop alignment [9]. From this treatment, general validity tests have been derived. This enables compound transformations to be applied in one go rather than step by step [17]. In [8], the class of transformations is further extended and presented in a unifying framework based on schedules.

Loop transformations, however, are not the only program transformations of interest when optimising programs for parallel execution. In this paper we describe a unifying framework for data transformations, similar in approach to the framework used for loop transformations. Data transformations can be loosely defined as those transformations concerned with the layout, storage and access of array data, rather than reordering the program control flow. Data transformations have a number of applications, including global index reordering, data alignment, data partitioning, access normalisation, false sharing elimination and storage reduction. For instance, in [16], copying of data is used to eliminate cache conflicts and in [7] the use of data transformations to reduce false sharing in parallel C programs is explored. In [4, 12, 15], the relative alignment of arrays to reduce communication overhead is examined. Data transformations can be used in those cases where data dependence prevents loop transformations and where access to other arrays should not be altered. They are unaffected by control-flow structure and may be used in the presence of imperfectly nested loops and arbitrary control-flow graphs. For a good survey of loop and data transformations, see [10].

Despite the wide applicability of such transformations, there has been, until recently, little work done to incorporate them into one single framework. In [6], the effect of row and column-major storage of arrays combined with loop transformations is investigated. Many of the points raised here are discussed, such as data layout being independent of (flow) dependences and how loop and data transformations are related. Their work, however, is preliminary in that the only data transformation discussed is *linearisation*, the mapping of n-dimensional arrays to 1-dimension. However, the loop transformations are full rank and dimension preserving. Hence there is an asymmetry in the treatment of loop and data transformations, weakening the unification theme in the paper. The transformations presented in this paper allow the row to column-major reordering described in [6] and describe a much larger set of transformations.

In [1] it is proposed that arrays be viewed as polyhedrons and that strip-mining and unimodular transformations be used for changing array layout, though they restrict their attention to permutations. In this paper we provide a general framework for data transformations, encompassing unimodular transformations. We show that this wide class of transformations is useful by giving an algorithm for constructing a data transformation that improves spatial locality and that leads us beyond simple permutations.

This paper makes the following contributions: it

- Describes a unifying framework for a large class of data transformations.
- Shows how existing transformations may be expressed within the framework.
- Defines the application of data transformations on program structure and code generation.
- Describes the symmetry between non-singular loop and data transformations.

- Provides sample algorithms to constructively generate data transformation and show their applicability in program optimisation.

The paper is organised as follows. In the next section a motivating example is presented. Section 3 defines the notation used throughout the paper, while section 4 defines the form of transformations. Section 5 presents validity criteria and section 6 describes how data and loop transformations can be integrated. Section 7 describes two applications of data transformations and finally, section 8 concludes this paper.

2 Example

In this section, a simple example is used to illustrate the effect of data transformations on programs and why they are useful for compiler optimisation. Stride 1 access of arrays is important in architectures with some form of cache structure as it allows exploitation of spatial locality. Loop interchange is frequently used to ensure this for simple access functions. For general affine accesses, a non-singular loop transformation [14] to first normalise the access is required, followed by loop interchange [13].

Loop transformations that improve spatial locality on one access may adversely affect others and even then may not be valid due to data dependences. The example in column 1 of figure 1 illustrates both these limitations with loop transformations. Fortran stores arrays in a column-major fashion. This is highlighted by the vertical lines within the diagrams of arrays `A` and `B`. The accessed region for each array is shaded grey, with arrows representing the traversal order imposed by the loops. At present there is bad stride access on both arrays `A` and `B`. Loop interchange would improve the access to `A` but is illegal due to the dependence vector $[1,-1]$. Furthermore, while access to `A` would improve, the access stride of `B` would still be poor. Changing column to row-major storage, as described in [6] and shown in column 2, would improve stride access to `A` but would again be of no use in improving access to `B`. This can be readily seen in the diagram of column 2.

Non-singular data transformations can be used as they are not restricted by data dependences, and they do not affect other array accesses within that loop nest. Although they must be globally propagated, this impact can be reduced as described in section 6.

Applying the appropriate data transformations gives the new program in column 3. Since arrays in Fortran need to be rectangular, the new `B` array is larger than the original. We have to take the bounding box to generate the new array bounds. The dark region represents unaccessed elements introduced by the need to restrict arrays to rectangular form. The skewed access region is mapped to a square one, and stride 1 access is now possible for access to both arrays. The form of the non-singular data transformation, how it is deduced, applied and its impact on actual machine performance is described later in section 7.1.

3 Notation

In this section we briefly describe the notation used to develop the data transformation framework. It is based on a linear algebraic representation of program constructs.

3.1 Iteration Spaces

The iterators in a Fortran program surrounding any statement can be represented as an $m \times 1$ column vector

$$J = [j_1, j_2, \ldots, j_m]^T$$

where m is the number of enclosing loops or iterators. The loop range or affine bounds of the iterators can be described by a system of inequalities defining the *polyhedron*

$$AJ \leq b \quad (1)$$

where A is a $(\ell \times m)$ integer matrix and b an $(\ell \times 1)$ vector. The integer values taken on by J define the *iteration space* of the iterators.

3.2 Index Space

The data storage of an array `A` can also be viewed as a rectangular polyhedron. We introduce *formal indices* $\mathcal{I}$ for the array to describe the array index space

$$\mathcal{I} = [i_1, i_2, \ldots, i_{\mathcal{D}(\mathtt{A})}]^T$$

where $\mathcal{D}(\mathtt{A})$ is the dimension of array `A`. The formal indices have a certain range which describe the size of the array, or *index space*, as follows:

$$\lambda \leq \mathcal{I} \leq \mu \quad (2)$$

where $\lambda = [\lambda_1, \ldots, \lambda_{\mathcal{D}(\mathtt{A})}]^T$ and $\mu = [\mu_1, \ldots, \mu_{\mathcal{D}(\mathtt{A})}]^T$ are $\mathcal{D}(\mathtt{A}) \times 1$ vectors. Due to the restrictions on the bounds of arrays in Fortran (or other languages), only *constant vectors* are required to describe array bounds.

3.3 Array Accesses

The subscripts in a reference to an array `A` represent a function that maps the values of the iteration space to the index space. If J is the iteration vector, we assume in this paper that these subscripts can be written as an affine mapping that has the following form

$$\mathcal{U}J + u$$

where $\mathcal{U}$ is a $\mathcal{D}(\mathtt{A}) \times m$ matrix and u is a $\mathcal{D}(\mathtt{A}) \times 1$ vector.

Thus the access to `B` in figure 1, `B(i+j,j)`, is denoted by

$$\mathcal{U}J + u = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The subscripts in a reference to an array are normally referred to by the *access matrix/vector pair* $(\mathcal{U}, u)$.

4 Data Transformations

This section describes the application of data transformations and the range of transformations contained with the class of non-singular matrices. Data transformations on an array update all access to that array and also change the way the data is stored.

Definition *A* data transformation, $(\mathcal{A}, a)$, *for an array* `A` *of dimension* n *consists of a (non-singular)* $n \times n$ *matrix* $\mathcal{A}$, *and a* $n \times 1$ *vector* a

When the shift vector a equals $\vec{0}$, we write $(\mathcal{A}, a)$ simply as $\mathcal{A}$. Given an access matrix/vector pair $(\mathcal{U}, u)$ for an array

Original	Column $\mapsto$ Row Major	Non-Singular Data Transformation

`Real A(4,8), B(8,4)`
`.`
`.`
`Do i = 1,3`
`  Do j = 2,4`
`    A(i,j) = A(i+1,j-1) + B(i+j,j)`
`  Enddo`
`Enddo`
`.`
`.`
	`Real A(4,8), B(8,4)`
	`C Row-major ordering`
	`.`
	`Do i = 1,3`
	`  Do j = 2,4`
	`    A(i,j) = A(i+1,j-1) + B(i+j,j)`
	`  Enddo`
	`Enddo`
	`.`
	`.`
		`Real A(8,4), B(4,11)`
		`.`
		`.`
		`Do i = 1,3`
		`  Do j = 2,4`
		`    A(j,i) = A(j-1,i+1) + B(j,i+4)`
		`  Enddo`
		`Enddo`
		`.`
		`.`

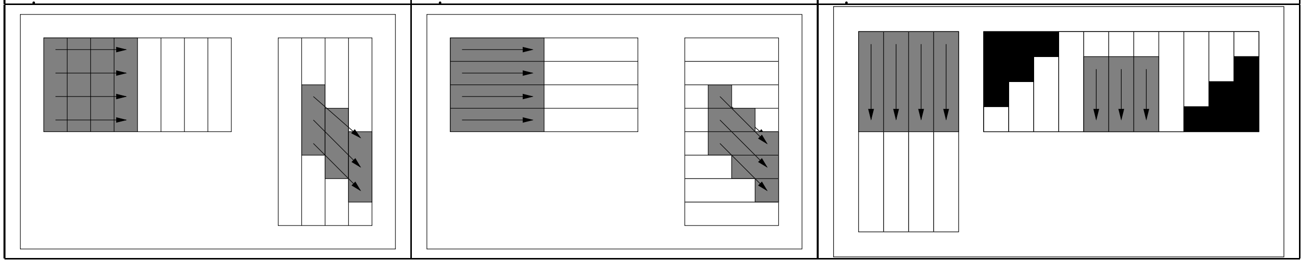

Figure 1: Stride Access Patterns

A occurring in the program, application of a data transformation $(\mathcal{A}, a)$ to this access results in a new access $(\mathcal{U}', u')$ given by

$$(\mathcal{U}', u') = (\mathcal{A}\mathcal{U}, \mathcal{A}u + a) \tag{3}$$

$\mathcal{A}$ is a **left-hand** acting transformation, in contrast to loop transformations which are **right-hand** acting. Unlike loop transformations, which are local to a loop nest but affect all contained references, data transformations affect just one array but affect all accesses to the array globally. Thus the transformation $(\mathcal{A}, a)$ must be applied to every occurrence of array A throughout the program. In addition the actual storage of the array must be updated. Thus the index space $\mathcal{I}$ of array A must also be updated.

The application of $(\mathcal{A}, a)$ is straightforward: all accesses are updated according to equation (3) and the index space is updated as described in section 4.1 below. The validity of such transformations is discussed in section 5. No change need be made to the loop or control structure of the program.

In the example in figure 1 the following transformations are applied to the arrays in column 1 to give the transformed form shown in column 3.

$$\mathcal{A}_{\mathtt{A}} : \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, a_{\mathtt{A}} : \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ and } \mathcal{A}_{\mathtt{B}} : \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} a_{\mathtt{B}} : \begin{bmatrix} 0 \\ 4 \end{bmatrix} \tag{4}$$

In section 7.1, a polynomial algorithm is presented which determines the correct transformation to obtain stride-1 access.

4.1 Transforming the Index Space

As well as transforming the accesses to an array, the actual storage for that array needs to be modified. The index space is transformed as follows.

Suppose $(\mathcal{A}, a)$ is a data transformation for an array A we need to compute a new index space $\lambda' \leq \mathcal{I}' \leq \mu'$. The new indices $\mathcal{I}'$ are related to the old indices $\mathcal{I}$ as

$$\mathcal{I}' = \mathcal{A}\mathcal{I} + a, \mathcal{I} = \mathcal{A}^{-1}\mathcal{I}' - \mathcal{A}^{-1}a$$

Substituting in equation (2), we obtain

$$\lambda + \mathcal{A}^{-1}a \leq \mathcal{A}^{-1}\mathcal{I}' \leq \mu + \mathcal{A}^{-1}a$$

If $\mathcal{A}$ is solely a permutation matrix, this is sufficient. However, in the general case further work is necessary: we must solve for $\mathcal{I}'$ and take the bounding box of the transformed index space, as arrays in Fortran have to be rectangular. To determine the new array bounds, Fourier-Motzkin elimination is applied to each index to find its range. That is, for an index we project away more outer indices to obtain the minimum and maximum values of an index.

For example, consider the access to array B in figure 1. $\mathcal{A}_{\mathtt{B}}$ and $a_{\mathtt{B}}$ are given in equation (4) and

$$\mathcal{A}_{\mathtt{B}}^{-1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

This gives the new bounds

$$\begin{array}{rcccl} 1 & \leq & i'_i & \leq & 4 \\ 5 - i'_i & \leq & i'_2 & \leq & 12 - i'_1 \end{array}$$

After applying Fourier-Motzkin elimination we obtain the new bounds `B(1:4,1:11)`.

4.2 Forms of $\mathcal{A}$

Within this paper we restrict our attention to non-singular forms of $\mathcal{A}$. This is analogous to concentrating on non-singular loop transformations. It follows that $\mathcal{A}$ always has an inverse though this inverse is not necessarily integer. For the purposes of showing the usefulness of such transformations, we can subdivide them and illustrate their effect on the program segment in figure 2. Figure 3 summarises the forms of the transformations considered.

Unimodular Unimodular matrices have been extensively studied and their properties will not be expounded her. They encompass *permutation*, *reversal* and *skewing* of the access matrix. Uses of such transformations include global index reordering and restricted forms of data alignment. Column 1 of figure 3 summarises the effect of unimodular transformations.

```
Do i = 1,n
 Do j = 2,n
  A(2*i,i+j) = i+j
Do i = 1,n
 Do j = 2,n
  A(2*i,j) = B(i,j)
```

Figure 2: Example

Integer determinant > 1 These matrices are a generalisation of unimodular matrices incorporating *scaling* or expansion. One use would be to expand small arrays so that when partitioned over a large template, the array is evenly distributed across all the processors. Scaling by cache-line size may also help alleviate false sharing. Scaling the first index by a factor of 2 is shown in figure 3.

Rational determinant Rational non-singular transformations allow *compression* of the access and hence of data space. This is useful when space is critical, especially for reducing the sizes of "windows" [?] in caches. Regular sparse accesses can be compressed so as to reduce the overall data allocation for that array. The example in the table shows compression in the first index by a factor of 2.

Shift This is a very simple transformation where the arrays are *shifted* by a constant value. This can prove useful when concerned with aligning data on coherency unit boundaries or to remove problems associated with cache hardware mapping. The effect of a shift of -1 is shown in figure 3.

4.3 Existing Transformations

While there has been much effort to show that loop transformations can be unified, there has been little work with regard to data transformations. Several existing transformations can be considered special cases of non-singular data transformations.
Global index reordering of a particular array corresponds to a simple permutation data transformation matrix.
Padding array declarations to avoid cache conflicts corresponds to an identity data transformation plus a shift vector.
Data Partitioning is in fact a rank deficient data transformation matrix[1] whose non-null rows correspond to partitioned array dimensions.
Data Alignment reorientates data with respect to a template. Non-singular data transformations cover all the linear alignment directives used in HPF.

All data layout schemes can be described as data transformations. Thus transformations from a wide range of applications can be viewed within one framework.

5 Validity

This section describes when data transformations can be legally applied. Data storage transformations can be considered to be an advanced form of renaming variables. Rather than just renaming a variable, e.g., x becomes y, we rename the index values as well. For example, `A(1,2)` will now become `B(3,4)`. Renaming or changing the symbol of a variable is legal as long as the renaming occurs for all occurrences of the variable and there does not exist a variable with the same name. The legality tests for $(\mathcal{A}, a)$ are similar.

[1] the legality of which is not considered in this paper

First of all, we must guarantee that all references are updated and secondly, that no clashes between two references occur. As an additional check we should ensure that all data dependences are preserved. In fact, it will be shown that non-singular data transformations always preserve data dependence. Fortran allows arrays to change dimension throughout a program especially when passing across subroutine boundaries. We assume that such a change in dimension does not occur in the program under consideration.

Let R be the total number of references to some array `A`. Let, for $1 \leq i \leq R$, the ith reference to `A` be given by access matrix $\mathcal{U}_i$ and vector u_i. Let $(\mathcal{A}, a)$ be a data transformation for `A`. All references are updated, that is, for all $1 \leq i \leq R$,

$$\mathcal{U}_i \mapsto \mathcal{A}\mathcal{U}_i \text{ and } u_i \mapsto \mathcal{A}u_i + a$$

The first requirement we impose on the data transformation matrix $\mathcal{A}$ is that after application array index functions map iterators to integer values. That is, for every point J in the iteration space,

$$\mathcal{A}\mathcal{U}_i J + \mathcal{A}u_i + a \in Z^{\mathcal{R}(\mathcal{A})}$$

where $\mathcal{R}(\mathcal{A})$ denotes the number of rows in $\mathcal{A}$.

The second requirement on $\mathcal{A}$ is that all distinct elements referenced in the original program are mapped to distinct points. Let $(\mathcal{U}_i, u_i)$ be an access matrix/vector pair occurring in a loop J_i and let $(\mathcal{U}_k, u_k)$ be an access matrix/vector pair occurring in another loop J_k. Then we require that for every iteration J_i and J_k in these loops,

$$\begin{aligned} &\mathcal{U}_i J_i + u_i = \mathcal{U}_k J_k + u_k \\ &\quad \text{iff} \quad \mathcal{A}\mathcal{U}_i J_i + \mathcal{A}u_i + a = \mathcal{A}\mathcal{U}_k J_k + \mathcal{A}u_k + a \end{aligned}$$

Note that we do not require that every element in the array is mapped to a distinct element, but rather that every element *referenced in the program* is mapped to a distinct element. This distinction is important since we want to consider storage compression for arrays that are only partially filled. Since in this paper we are only considering data transformation matrices that are non-singular and hence bijective, this requirement is automatically fulfilled. In future research we hope to consider singular data transformation matrices also. Next we consider the validity of non-singular data transformations. First, we state a lemma the proof of which is trivial since $\mathcal{A}$ is bijective.

Lemma 1 *Let $(\mathcal{A}, a)$ be a non-singular data transformation for some array* `A`. *Suppose $(\mathcal{U}_1, u_1)$ and $(\mathcal{U}_2, u_2)$ be two access matrix/vector pairs for* `A` *in some loop. Then there exists a dependence from iteration J_1 to iteration J_2 caused by these accesses before the application of $(\mathcal{A}, a)$ to* `A` *if and only if there exists a dependence from J_1 to J_2 after the application of $(\mathcal{A}, a)$ to* `A`.

This Lemma immediately leads to the following proposition.

Proposition 2 *The application of a non-singular data transformation is always valid.*

6 Unifying Loop and Data Transformations

There has been references to and parallels drawn with loop transformations within this paper. In fact, there is a strong symmetry between data and loop transformations. This can be readily seen if we look at the effect of either transformation from an *access* rather than a loop or data storage point of view.

$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ permutation	`Do i = 1,n`
	` Do j = 2,n`
	`  a(i+j,2*i) = i+j`
	`Do i = 1,n`
	` Do j = 2,n`
	`  a(j,2*i) = b(i,j)`
		$\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ scaling	`Do i = 1,n`
			` Do j = 2,n`
			`  a(2*i,2*i+2*j) = i+j`
			`Do i = 1,n`
			` Do j = 2,n`
			`  a(2*i,2*j) = b(i,j)`

$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ reversal	`Do i = 1,n`
	` Do j = 2,n`
	`  a(2*i,-i-j) = i+j`
	`Do i = 1,n`
	` Do j = 2,n`
	`  a(2*i,-j) = b(i,j)`
		$\begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ compression	`Do i = 1,n`
			` Do j = 2,n`
			`  a(i,i+j) = i+j`
			`Do i = 1,n`
			` Do j = 2,n`
			`  a(i,j) = b(i,j)`

$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ skewing	`Do i = 1,n`
	` Do j = 2,n`
	`  a(2*i,3*i+j) = i+j`
	`Do i = 1,n`
	` Do j = 2,n`
	`  a(2*i,2*i+j) = b(i,j)`
		$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ shifting	`Do i = 1,n`
			` Do j = 2,n`
			`  a(2*i,i+j-1) = i+j`
			`Do i = 1,n`
			` Do j = 2,n`
			`  a(2*i,j-1) = b(i,j)`

Figure 3: Transformations

First, let us recapitulate some of the issues involved. For the sake of clarity, we ignore shifts in the discussion below. Given a loop L and an array A, we denote the iteration space of L by $S(L)$ and the index space of A by $S(\mathtt{A})$. Then an access can be seen as mapping $(\mathcal{U}, u) : S(L) \to S(\mathtt{A})$. If we apply a transformation $\mathcal{T}$ to L we obtain a new loop L', and we can view the transformation as a mapping $\mathcal{T} : S(L) \to S(L')$. On the other hand, if we apply a transformation $\mathcal{A}$ to A we obtain a new array A', and we can view the transformation $\mathcal{A} : S(\mathtt{A}) \to S(\mathtt{A}')$. In order to obtain the bounds for L' and A', we need to use $\mathcal{T}^{-1}$ and $\mathcal{A}^{-1}$, respectively, for expressing the new iterators or indices in terms of the old iterators and indices, respectively. However, for transforming accesses, in case of a loop transformation we need to go from $S(L')$ to $S(L)$, using the inverse of the loop transformation, $\mathcal{T}^{-1}$, and then to $S(\mathtt{A})$, using the access matrix. Hence $\mathcal{T}^{-1}$ is a *right hand operating* transformation. In case of a data transformation, we need to go from $S(L)$ to $S(\mathtt{A})$, using the access matrix, and then to $S(\mathtt{A}')$, using the data transformation $\mathcal{A}$. So $\mathcal{A}$ is a *left hand operating* transformation. Thus a loop and a data transformation together have the following effect upon a data access matrix:

$$\mathcal{U} \mapsto \mathcal{A}\mathcal{U}\mathcal{T}^{-1}$$

Frequently, both transformations can have the same effect on one particular access. In such a situation the choice of a data or loop transformation will depend on legality criteria being satisfied and any side effect on the rest of the program.

One very useful outcome of these observations is that the *global* impact of data transformations can frequently be resolved *locally* by loop transformations. Consider the example in figure 1. The initialisation of B is not shown here. Let us assume it is of the form shown in column 1 of figure 4. On propagating the alignment transformation the initialisation code will be transformed into that of column 2. Unfortunately this has very poor access stride, so we have shifted the problem from one place to another. However, the access stride can be improved locally by a loop transformation ($\mathcal{T} = \mathcal{U}^{-1}\mathcal{A}\mathcal{U}$ and $t = a$) to give the code in column 3. In this case there are no problems due to data dependence and conflicting access requirements.

By considering non-singular loop and data transformations at the same time, we can apply both elementary row and column operations to an access. Further investigation of the relationship between loop and data transformations is beyond the scope of this paper and will be the subject of future work.

7 Applications

In this section the use of data transformations in a number of applications is outlined. Each subsection will describe the optimisation goal and a constructive algorithm to generate the appropriate transformation. Experimental results on the SGI Challenge and KSR-1 are presented to illustrate the usefulness of data transformations by comparing them to other techniques.

7.1 Spatial Locality

Stride-1 access of arrays is important in architectures with some form of cache structure since this allows exploitation of spatial locality. Assuming a column-major ordering of arrays, the innermost iterator should ideally access only the first index, if any. Accessing the second or subsequent index should be avoided as it will destroy spatial locality. Assuming that the dimension of the referenced array is n and that the nesting depth of the surrounding loops is m, the desired structure of an access matrix $\mathcal{U}$ for spatial locality is therefore

$$\mathcal{U} = \begin{bmatrix} y & 1 & o_1 \\ Y & o_2 & O \end{bmatrix} \tag{5}$$

where y is a $1 \times \ell$ row matrix, Y an $(n-1) \times \ell$ matrix, o_1 an $1 \times (m-\ell-1)$ null row matrix, o_2 an $(n-1) \times 1$ null column matrix and O a $(n-1) \times (m-\ell-1)$ null matrix. The 1 is the right-most non-null entry in row one and corresponds to the most inner iterator referenced by the first index. For good spatial locality, that iterator must not be referred to by any other index, i.e., there should be just one column entry for that iterator. This implies that o_2 must be null. More than one entry will imply diagonal or skewed access.

For example, assuming an iterator ordering k,j,i, then A(i,j+k) and B(i+j,j) are good accesses, while C(i,i+j) is not. This can be seen from their access matrices:

$$\mathcal{U}_{\mathtt{A}} : \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad \mathcal{U}_{\mathtt{B}} : \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \mathcal{U}_{\mathtt{C}} : \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

where only the access matrix of C does not have the desired structure. In both A(i,j+k) and B(i+j,j) the innermost iterator i is only referenced in the first index while the access C(i,i+j) refers to i in both indices.

<table>
<tr><th>Before Data Transformation</th><th>After Data Transformation</th><th>After Loop Transformation</th></tr>
<tr><td><pre>Real B(8,4)
.
.
Do j = 1,4
  Do i = 1,8
    B(i,j) = i+j
  Enddo
Enddo
.
.</pre></td><td><pre>Real B(4,11)
.
.
Do j = 1,4
  Do i = 1,8
    B(j,i-j+4) = i+j
  Enddo
Enddo
.
.</pre></td><td><pre>Real B(4,11)
.
.
Do j = 1,11
  Do i = max(1,5-j), min(4,12-j)
    B(i,j) = 2*i +j -4
  Enddo
Enddo
.
.</pre></td></tr>
<tr><td>Original</td><td>Apply $\mathcal{A} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}$ and $a = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$</td><td>Apply $\mathcal{T} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$ and $t = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$</td></tr>
</table>

Figure 4: Interaction of Loop and Data Transformations

7.1.1 Algorithm

To achieve the desired structure of the access matrix requires reordering of the matrix by elementary row or column operations which directly correspond to left and right hand transformations, i.e., data and loop transformations. The algorithm in figure 5 describes a constructive technique for optimising spatial locality using data transformations. It attempts to create a matrix with an anti-diagonal structure using rowwise operations. Such a matrix has the desired property of equation (5).

1. Let T be the reverse permutation of identity, where $t_{ij} = 1$ if $j - i = n - 1$, 0 otherwise.
2. Apply rowwise operations $\mathcal{A}$ to reduce $\mathcal{U}T$ to integer row echelon form.
3. Determine lower bounds λ_k and upper bounds μ_k of each index i_k after applying $(\mathcal{A}, 0)$, using Fourier-Motzkin elimination.
4. Let $a_k = -\lambda_k + 1$ for all k, yielding the shift vector a. This shift vector guarantees that all lower bounds become 1, as is the default in Fortran.
5. Apply the data transformation $(\mathcal{A}, a)$ to the array.

Figure 5: Spatial Locality Algorithm

To illustrate this algorithm consider the access to `B` in figure 1 and assume that the array `B` is of size $(2n \times n)$. Applying the algorithm, we obtain the initial data transformation

$$\mathcal{A} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}$$

This gives the following lower and upper bounds after applying $\mathcal{A}$:

$$1 \le i_1 \le n \quad \text{and} \quad -n+1 \le i_2 \le 2n-1$$

The new lower bounds of the array are negative after applying $\mathcal{A}$, so the shift vector a is applied to restore the indices to positive values, where $a = [0, n]^T$. The updated program for $n = 4$ is given in column 3 of figure 1.

This algorithm is applicable to more complex references such as `A(i+j+k,i+k,i-j+k)`. If the above algorithm is applied we have `A(i+j+k,i+k,i-j+k)` $\mapsto$ `A(2*i+k,j,0)`, where the innermost iterator `k` has stride 1 access on `A`. Although such references are unlikely to be as complex in practice, it demonstrates the generality of the above scheme.

Rank Deficient Accesses If data transformations to improve spatial locality on a particular access are restricted to permutations then they can be seen as a formulation of global index reordering or a change from row to column-major storage. In the general case for arbitrary affine accesses they can be seen as the data dual of access normalisation [13]. Data transformations can be shown to be strictly more powerful than loop transformations in enhancing spatial locality, as there exists accesses, namely rank deficient accesses, where no loop transformation can improve its spatial locality, but a data transformation can.

Rank deficient accesses, where two rows are dependent, cannot be removed by column operations (loop transformations). This causes a problem if the two dependent rows contain an entry in the innermost iterator column.

For example, consider the access `A(i+j,i+j)`. A loop transformation may map

`A(i+j,i+j)` $\mapsto$ `A(i',i')`

while a data transformation may have the following effect:

`A(i+j,i+j)` $\mapsto$ `A'(i+j,0)`

In this case data transformations can remove diagonal access causing references which cannot be removed by loop transformations. Column/row-major reordering is also unable to remove the poor stride access in rank deficient accesses.

7.1.2 Results

To illustrate the benefit of the above algorithm the three versions of the program shown in figure 1 were executed and the results shown in figure 6 on one processor of an SGI Challenge. Parallel versions could not be evaluated due to the serialising data dependences within the program[2]. Changing from column to row-major layout provides a small improvement on large data sizes as there is improved access to array `A`. The poor stride on `B`, however, prevents significant performance improvement. The data transformation approach, based on the algorithm in figure 5, improves the stride access to both arrays and shows up to a factor of 5 improvement over the existing methods.

[2] The figure shows time against varying data sizes for three layouts, rather then temporal performance (1/Time) against processor number which is used in the other (parallel) applications.

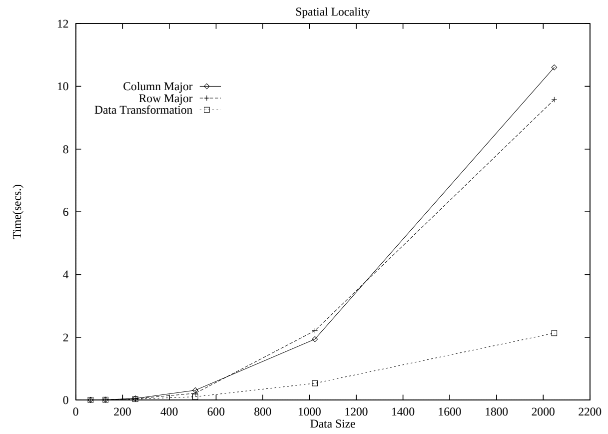

Figure 6: Spatial Locality

7.2 Page Replication

In this section we will look at page replication in Cache-only Memory Architectures (COMAs). Let the dimension of the first array index be n, the number of processors p and the size of the coherence unit[3] c. Difficulties arise in COMAs when $\frac{n}{p} > c$ for 2- or higher dimensional arrays, if partitioned on the first index. This means that every processor is allocated a collection of rows of the array. Data is allocated column-wise to pages and each time an element is referenced, the page must be first allocated in memory by the Operating System before it may be accessed. Partitioning on the first index of a 2-dimensional array implies that potentially every column will be accessed by every processor and thus all pages must be allocated in each processors attraction memory. For large arrays, there will be extensive cache conflict and performance degradation. However, in those cases where parallelism is only available in the first array index, existing loop based approaches cannot alleviate this problem without loss of parallelism.

7.2.1 Algorithm

```
Real A(200,400), B(200,400)
Do k = 1,n
  Do i = 1,n
    A(i,k) = A(i,k-1) + B(i,k)
  Enddo
Enddo
```

Figure 7: ADI fragment: before transformation

Consider the program fragment from the ADI example [5] shown in figure 7. Here parallelism is available only in the first index. Interchanging loops will not change this but will actually lead to worse stride access. If, however, the storage of all arrays were transposed, leading to a columnwise data partitioning throughout the program, then page replication is avoided. However, this is at the expense of stride access as the innermost iterator now strides through the second array index. This is easily rectified by loop interchange and is shown in figure 8.

The algorithm in figure 9 performs a data storage transformation program wide if partitioning is upon the first index. It takes as input a partitioning matrix $\mathcal{P}$. With each

[3]Cache line, page or, in the case of the KSR, sub-page.

```
Real A(400,200), B(400,200)
Do i = 1,n
  Do k = 1,n
    A(k,i) = A(k-1,i) + B(k,i)
  Enddo
Enddo
```

Figure 8: ADI fragment: after transformation

partitioning matrix $\mathcal{P}$ a serialising matrix $\mathcal{S}$ is associated such that

$$\mathcal{P} + \mathcal{S} = I$$

Given the formal indices $\mathcal{I}$, $\mathcal{PI}$ and $\mathcal{SI}$ define those indices to be partitioned and serialised, respectively. A full rank $\mathcal{P}$ implies that all the indices will be partitioned; a null matrix means the array is not distributed.

The partitioning matrix $\mathcal{P}$ is checked to see if it partitions the first index and if so, whether there are other indices which could be partitioned instead. This can only be true if $\mathcal{P}$ is not full rank. If successful, the partitioning matrix, all array accesses and data storage are updated. In the algorithm $E_{1,j}$ denotes the identity matrix I with columns 1 and j interchanged. Its application swaps the first row of a matrix with the jth, hence the jth row will now be partitioned rather than the first.

if $\mathcal{P}$ is not full rank and $\mathcal{PI}[1] \neq 0$
then
    Let j be the first null row of $\mathcal{P}$
    Let $\mathcal{A} = E_{1,j}$
    Apply $(\mathcal{A}, 0)$ to all accesses
    Let $\mathcal{P}' = \mathcal{A}\mathcal{P}\mathcal{A}^{-1}$
endif

Figure 9: Page Replication Algorithm

In the example given in figure 7, we have before transformation the following value of $\mathcal{P}$.

$$\mathcal{P} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Given the formal indices $\mathcal{I}$, $\mathcal{PI}$ defines those indices which will be partitioned. In this case, $\mathcal{PI}[1]$, the first element of $\mathcal{PI}$, equals i_1. Hence the first index is to be partitioned. $\mathcal{P}$ is not full rank as one of its rows is null; it is a 2-dimensional matrix of rank 1. If $\mathcal{P}$ were full rank, this would imply that all indices were to be partitioned and there would be no point in reordering indices. In the case of a rank deficient partition matrix, other indices can be chosen to partition.

As there is only one null row in $\mathcal{P}$, $j = 2$ and $\mathcal{A}$ is readily found:

$$\mathcal{A} = E_{1,2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Application of the data transformation $(\mathcal{A}, 0)$ gives the new accesses, index space and new partitioning matrix $\mathcal{P}'$. The new index to partition on is given by

$$\mathcal{P}'\mathcal{I}' = \begin{bmatrix} 0 \\ i_2' \end{bmatrix}$$

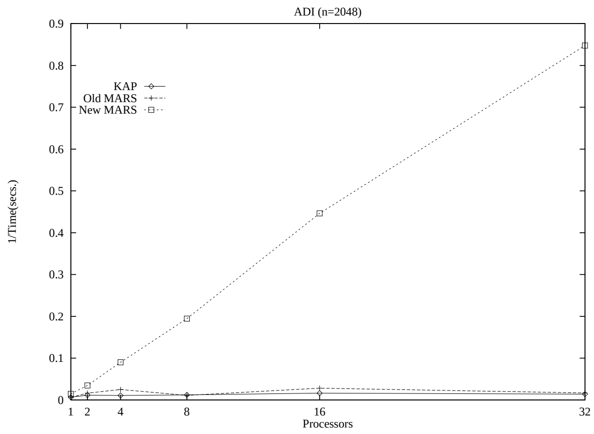

Figure 10: ADI fragment: n = 2048

which is the second index, avoiding the problem of false sharing and page replication[4]. After loop interchange is performed, the program in figure 8 is found.

This algorithm was implemented in the MARS compiler [3] after the data partitioning decision is made. Although it may also be necessary to update loop structures using loop interchange in cases where bad stride occurs, it is not necessary to include this in the present transformation, as there already is an existing module in MARS which reorders loop for locality after partitioning and mapping has occurred [3].

7.2.2 Results

To demonstrate the importance of this algorithm we compared it with the existing MARS compiler and KAP, an existing commercial auto-parallelising compiler. We applied KAP, MARS and the new MARS compiler in which this transformation is implemented to the ADI code fragment for the data size n=1024 and ran the resulting parallel programs on the KSR on 1, 2, 4, 8, 16 and 32 processors. The results are shown in figures 10. Applying the data transformation can give dramatic performance improvements by up to a factor of 50 improvement over the old version of MARS and a factor of up to 60 over KAP.

8 Conclusion

In this paper we have described the use of non-singular data transformations in optimising program performance. We have described their impact on program structure and developed validity tests for their application. Constructive algorithms have been developed to generate transformations that can significantly improve program performance outperforming existing techniques. Non-singular data transformations unify existing transformations and are strongly connected with non-singular loop transformations. Future work will investigate the connection between these classes of transformation and extend the class of transformations examined beyond non-singular matrices.

References

[1] J.M. Anderson S.P. Amarasinghe and M.S. Lam. **Data and Computation Transformations for Multiprocessors**, Proc. of Principles and Practice of Parallel Programming, ACM Press, 1995.

[2] U. Banerjee. **Unimodular Transformations of Double Loops**, Proc. of 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine CA, August 1990.

[3] F. Bodin and M.F.P. O'Boyle. **A Compiler Strategy for SVM** Third Workshop on Languages, Compilers and Runtime Systems, New York, Kluwer Press, May 1995.

[4] S. Chatterjee, J.R. Gilbert, R. Schreiber and S-H. Teng. **Automatic Array Alignment in Data Parallel Programs**, Proc. of Principles of Programming Languages, Charleston, ACM Press, January 1993.

[5] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka and C.-W. Tseng. **Unified Compilation of Fortran 77D and 90D**, ACM Letters on Programming Languages and Systems Vol2 Nos 1-4, March - December 1993.

[6] M. Cierniak and W. Li. **Unifying Data and Control Transformations for Distributed Shared-Memory Machines**, Programming Language Design and Implementation, June 1995.

[7] T.E.Jeremiassen and S.J. Eggers, **Reducing False Sharing on Shared Memory Multiprocessors through Compile-Time Data Transformations**, Proc. of Principles and Practice of Parallel Programming, ACM Press, 1995.

[8] W.A.Kelly, W.Pugh, **A Framework for Unifying Reordering Transformations** Tech Report CS-TR-3193, Dept CS, Univ. Maryland, May 1993.

[9] D. Kulkarni and M. Stumm. **CDA Loop Transformations**, Languages, Compilers and Run-Time Systems for Scalable Computers, Kluwer Academic Press, May 1995.

[10] D. Kulkarni and M. Stumm. **Loop and Data Transformations: A Tutorial** University of Toronto, Tech Rep CSRI-337, June 1993.

[11] P.M.W. Knijnenburg, E. Ayguadé and J. Torres. **Multi-transformations of Nested Loops for Parallelizing Compilers**, Tech. Rep. 96–14, Leiden University, 1996.

[12] J. Li and M. Chen. **Index Domain Alignment: Minimising Cost of Cross-Referencing between Distributed Arrays**, IEEE Proc. of the Third Symposium on the Frontiers of Massively Parallel Computation, 1990.

[13] W. Li and K. Pingali. **Access Normalisation: Loop Restructuring for NUMA Compilers**, Cornell University, Department of Computer Science, TR92-1278, April 1992.

[14] W. Li and K. Pingali. **A Singular Loop Transformation Framework Based on Non-singular Matrices**, Proc. of the Fifth Workshop on Languages and Compilers for Parallelism, August 1992.

[15] M.F.P. O'Boyle and G.A. Hedayat. **Data Alignment: Transformations to Reduce Communication on Distributed Memory Architectures**, Proc. of Scalable High Performance Computing Conference, Williamsburg, *IEEE Press*, April 1992.

[16] O.Temam, E.D.Granston and W.Jalby. **To Copy or Not To Copy: A compile-time technique for assessing when data copying should be used**, Proc. of Supercomputing '93, Portland Oregon, Nov. 1993.

[17] M.E. Wolf and M. Lam. **An Algorithmic Approach to Compound Loop Transformations**, in *Advances in Languages and Compilers for Parallel Processing*, A. Nicolau, D. Gelernter, T. Gross and D. Padua (eds.), Pitman London, 1991.

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

[4] The local node programs after partitioning are not shown here.