Integrating Loop and Data Transformations for Global Optimisation

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 is concerned with integrating global data transformations and local loop transformations in order to minimise overhead on distributed shared memory machines such as the SGi Origin 2000. By first developing an extended algebraic transformation framework, a new technique to allow the static application of global data transformations, such as partitioning, to reshaped arrays is presented, eliminating the need for expensive temporary copies and hence eliminating any communication and synchronisation. In addition, by integrating loop and data transformations, any introduced poor spatial locality and expensive array subscripts can be eliminated. A specific performance improving algorithm is implemented giving significant improvements in execution time.

1. Introduction

In order to achieve acceptable performance on current distributed shared memory machines, it is essential to exploit the available parallelism, make efficient use of the memory hierarchy and minimise non-local memory accesses, load imbalance and synchronisation overhead. Typically, a loop based parallelisation approach [13] is used which is local in nature, as each loop nest is separately examined and parallelised. Although each loop nest in isolation may perform well, they may perform poorly when combined due to significant communication and synchronisation between loop nests.

Another approach is to consider data orientated parallelisation [8], traditionally developed for distributed memory compilation but also used for distributed shared memory [1, 6]. This approach is primarily concerned with mapping arrays to processors and has a global, program wide, effect. It tries to globally trade off costs for an entire program, in contrast to loop based approaches. However, this global approach breaks down when a particular array has two different layouts, for instance when it is reshaped at a subroutine boundary. The partitioning of the original array and its reshaped instance must be consistent as they refer to the same actual array elements. At present, there is no efficient static means to apply data transformations such as partitioning to reshaped arrays. Expensive temporary copies must normally be made at run-time on entry to the procedure and restored on exit [5], introducing additional communication and synchronisation.

Another problem associated with global data transformations is that in determining a parallelisation that is globally acceptable, it may have an adverse affect on the performance of a particular statement within a loop nest. For instance, data alignment may transpose the layout of one array, reducing overall communication between processors, but in certain loop nests cause poor stride access through the local data cache by destroying spatial locality.

These two problems of data parallelisation, namely, reshaped arrays and the adverse effect on local loop nests, are the subject of this paper. By developing an extended framework for loop and data transformations, we have developed a technique to statically determine the data layout for reshaped arrays, eliminating the need for temporary copies and the associated communication and synchronisation overhead. To solve the second problem of the (potential) local adverse effects of global data transformations on loop nests, loop transformations can be used to undo these adverse effects.

In order to optimise programs globally, it is essential that a compiler is able to combine loop and data transformations. Recent work [1, 4, 7, 12] has focussed on combining the loop based parallelisation with data layout approaches [8] in order to trade-off these conflicting requirements. However, these approaches are restricted in that transformations, including partitioning, strip-mining and linearisation, cannot be directly incorporated within their representation.

This paper develops a new approach to combining loop

Original Code (1)

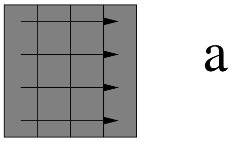
```
REAL A(0:3,0:3)
Do i = 0, 2
  Do j = 1,3
    A(i,j) = A(i+1,j-1)
           + D(j,i)
  Enddo
Enddo

call Reshape(a)
```

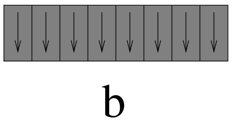
Reshaped Array (2)

```
Subroutine Reshape (B)
REAL B(0:1,0:7)
Do j = 0, 7
  Do i = 0,1
    B(i,j) = B(i,j) + 1
  Enddo
Enddo
```

Original Access (3)



Reshaped Access (4)



After Data Transformation(5)

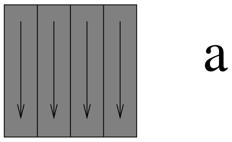
```
REAL A(0:3,0:3)
Do i = 0,2
  Do j = 1,3
    A(j,i) = A(j-1,i+1)
           + D(j,i)
  Enddo
Enddo

call Reshape(A)
```

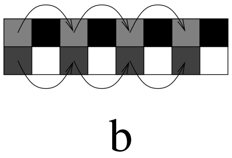
Propagated Transformation (6)

```
Subroutine Reshape (B)
REAL B(0:1,0:7)
Do j = 0, 7
  Do i = 0,1
    B(mod(4*mod((2*j+i),4)
      +(2*j+i)/4,2),
      (4* mod((2*j+i),4)
      +(2*j+i)/4)/2)  + = 1
  Enddo
Enddo
```

New Access (7)



New Reshaped Access (8)



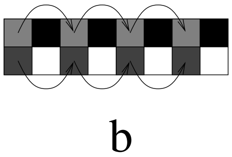
Loop Restructuring (9)

```
Subroutine Reshape (B)
REAL B(0:1,0:7)
Do j = 0, 3
  Do i = 0,3
    B(mod(j,2),2*i+j/2) += 1
  Enddo
Enddo
```

More Loop Restructuring (10)

```
Subroutine Reshape (B)
REAL B(0:1,0:7)
Do j1 = 0,1
 Do j2 = 0,1
  Do i = 0,3
    B(j2,2*i+j1) +=1
  Enddo
 Enddo
Enddo
```

Access Pattern(11)



Data Restructuring (12)

```
Subroutine Reshape (B)
REAL B(0:3,0:3)
Do j = 0, 7
  Do i = 0,1
    B(j/2,i+2*mod(j,2)) += 1
  Enddo
Enddo
```

Loop and Data Restructuring (13)

```
Subroutine Reshape (B)
REAL B(0:3,0:3)
Do j1 = 0,1
 Do i = 0,1
  Do j2 = 0,3
    B(j2,i+2*j1) += 1
   Enddo
  Enddo
 Enddo
Enddo
```

Access Pattern(14)

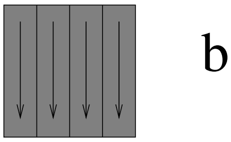


Figure 1. Transformations

and data transformations, introducing rank modifying transformations which allow generalised linearisation and strip-mining of loop and data spaces. Its main practical use is that it allows, for the first time, the static application of data transformations, such as global index reordering and data partitioning, to reshaped arrays. Applying global data transformations to reshaped arrays can, however, produce complex and inefficient code but, by integrating loop and data transformations, we can also systematically eliminate any complex array access function introduced. This dramatically improves spatial locality by restructuring data to have stride-1 access wherever possible.

This paper is organised as follows. In the next section, a motivating example showing the applicability of our integrated approach is presented. Section 3 presents the notation used within this paper and is followed by section 4 which describes the form and properties of our novel rank modifying transformations. Section 5 presents an algorithms and experimental results for data layout propagation, showing the significant improvement of such a scheme. Section 6 briefly reviews related work and is followed by some concluding remarks.

2. Example

In this section, we consider a simple example to illustrate the two problems tackled in this paper. Consider the program fragment in figure 1, box 1. In Fortran arrays are stored column-wise and therefore the reference to array `A` has a non stride-1 access pattern shown in box 3. The program fragment in box 2 contains references to `B` which are reshaped references to A[1] with perfect stride-1 access, as shown in box 4. Loop permutation to improve the access to array A in box 1 is not possible due to the data dependence and thus a compiler may wish to apply a data transformation as described in [12] to ensure stride-1 access. This is a simple permutation which gives the code in box 5 with the stride-1 access pattern shown in box 7. Its effect must be propagated to all accesses, including reshaped ones. Using the techniques described in section 5, we can obtain the code in box 6 and the access pattern shown in 8. It is clear that such access functions will be expensive, possibly outweighing any benefit to the improved stride access to A. Furthermore, this will not improve the pathological "leapfrog" access to B.

The second table in figure 1 shows different attempts to improve the loop structure and data access pattern of the code in box 6. Using a combination of strip-mining and linearisation, the loop can be transformed to that shown in box 9. The access pattern, shown in box 11, is unchanged, but the access function is considerably simplified. By the application of a further rank modifying loop transformation, we have the code in box 10, where all `mod` and `div` operators have been removed. This, however, does not affect the data access pattern shown in box 11. If instead a data transformation is applied to B, we have the code shown in box 12. Finally, if both loop and data transformations are combined, we obtain the code in box 13 which has stride-1 access shown in box 14. Thus, by a combination of rank modifying data and loop transformations, we have stride-1 access on all arrays without excessively expensive access functions. In section 5 we develop automatic techniques which select the appropriate transformations and show the impact of such restructuring on actual performance.

3. Notation

In this section, we briefly describe the notation used to describe those transformation used throughout the paper.

The loop indices, or *iterators*, 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. The loop ranges can be described by a system of inequalities defining the *polyhedron* $\mathbf{B}J \leq \mathbf{b}$. For ease of presentation, we assume that $\mathbf{B}$ is a $(2M \times M)$ integer matrix and $\mathbf{b}$ a $(2M \times 1)$ vector. The integer values taken on by J define the *iteration space* of the loop.

The data storage of an array A can also be viewed as a polyhedron. We introduce *formal indices* $\mathcal{I} = [i_1, i_2, \ldots, i_N]^T$, where N is the dimension of array A, to describe the *array index space*. This space is given by the polyhedron $\mathbf{A}\mathcal{I} \leq \mathbf{a}$, where $\mathbf{A}$ is a $(2N \times N)$ integer matrix and $\mathbf{a}$ a $(2N \times 1)$ vector. The integer values taken on by $\mathcal{I}$ define the index space of the array. In this paper we assume that all lower bounds of arrays are zero.

[1] Such aliasing can occur either due to equivalencing, or more usually, due to reshaping across subroutine boundaries.

```
Real A(0:3,0:7), B(0:3,0:7)

Do j = 0,3
 Do i = 0,3
  A(i,j) = A(j,i) + B(i,j)
 Enddo
Enddo
```

Figure 2. Example loop

We assume that the subscripts in a reference to an array A can be written as $\mathcal{U}J + u$, where $\mathcal{U}$ is a $(N \times M)$ matrix and u is a $(N \times 1)$ vector.

4. Rank Modifying Transformations

In this section, we first describe the form and properties of rank modifying transformations. This is followed by an illustrative example.

4.1. Data Transformations

A data transformation is applied to the index space of a particular array and *all* accesses to that array throughout the program and is therefore *global* in nature. A $(k \times N)$ linearisation matrix L is a transformation which maps an N dimensional index vector $\mathcal{I}$ to a new $k < N$ dimensional index vector $\mathcal{I}' = L\mathcal{I}$. Each array access $\mathcal{U}$ for an array A must be globally updated such that $\mathcal{U}' = L\mathcal{U}$. Data transformations are therefore *left-hand* transformations when applied to array access functions. The bounds of the new index space $\mathbf{A}'\mathcal{I}' \leq \mathbf{a}'$ must also be determined:

$$\mathbf{A}' = X\mathbf{A}L^{\dagger} \text{ and } \mathbf{a}' = X\mathbf{a} \tag{1}$$

and

$$X = \begin{bmatrix} L & 0 \\ 0 & L \end{bmatrix} \tag{2}$$

In equation (1), $L^{\dagger}$ is a transformation that is inverse to L on the index space of A. That is, for every index point $\mathcal{I}$ of A, $L^{\dagger}(L(\mathcal{I})) = \mathcal{I}$. We call such an inverse transformation $L^{\dagger}$ a *local inverse* for L on the index space of A.

Rank increasing transformations, such as strip mining, for which $k > N$ are denoted by S.

4.2. Loop Transformations

Loop transformations are applied to the iterators in a loop nest and to all accesses within a loop nest and are thus *local* in nature. A $(k \times M)$ linearisation matrix L is a transformation which maps an M dimensional iteration vector J

	Data	Loop	Combined

$\left[\begin{array}{cc} 1 & 4 \end{array}\right]$	`Real A(0:31), B(0:3,0:7)`
	
	`Do j = 0,3`
	` Do i = 0,3`
	`   A(i+4*j) = A(j+4*i) +B(i,j)`
	` Enddo`
	`Enddo`
		`Real A(0:3,0:7), B(0:3,0:7)`
		
		`Do i = 0,15`
		` A(mod(i,4),i/4) = A(i/4,mod(i,4))`
		`                 + B(mod(i,4),i/4)`
		`Enddo`
			`Real A(0:31), B(0:3,0:7)`
			
			`Do i = 0,15`
			` A(i) = A(i/4+4*(mod(i,4))`
			`      + B(mod(i,4)+4*(i/4))`
			`Enddo`

$\left[\begin{array}{c} (\cdot)\%2 \\ (\cdot)/2 \end{array}\right]$	`Real A(0:1,0:1,0:7), B(0:3,0:7)`
	
	`Do j = 0,3`
	` Do i = 0,3`
	`  A(mod(i,2),i/2,j) =`
	`   A(mod(j,2),j/2,i) + B(i,j)`
	` Enddo`
	`Enddo`
		`Real A(0:3,0:7), B(0:3,0:7)`
		
		`Do j = 0,3`
		` Do i2 = 0,1`
		`  Do i1 = 0,1`
		`    A(i1+2*i2,j) = A(j,i1+2*i2)`
		`             + B(i1+2*i2,j)`
		`  Enddo`
		` Enddo`
		`Enddo`
			`Real A(0:1,0:1,0:7), B(0:3,0:7)`
			
			`Do j2 = 0,1`
			` Do j1 = 0,1`
			`  Do i2 = 0,1`
			`   Do i1 = 0,1`
			`     A(i1,i2,j1+2*j2)`
			`           = A(j1,j2,i1+2*i2)`
			`           + B(i1+2*i2,j1+2*j2)`
			
			`   Enddo`
			`  Enddo`
			` Enddo`
			`Enddo`

Figure 3. Loop and Data Transformations

to a new $k < M$ dimensional iteration vector $J' = LJ$. Each access $\mathcal{U}$ within the loop nest must be updated such that $\mathcal{U}' = \mathcal{U}L^\dagger$. Thus, loop transformations are *right-hand* acting transformations when applied to array accesses. The new bounds of the new iteration space $\mathbf{B}'J' \leq \mathbf{b}'$ must be determined:

$$\mathbf{B}' = X\mathbf{B}L^\dagger \text{ and } \mathbf{b}' = X\mathbf{b} \quad (3)$$

and X is defined in equation (2).

4.3. Form of Transformations

In this paper, we restrict our attention to generalised strip-mining and linearisation. In order to describe such transformations in an algebraic framework, it is necessary that the transformation matrices may now include integer division and modulus operations as well as integers. If we need to include the operation "divide by n" as an entry in a matrix, we will write this entry as $(\cdot)/n$. Likewise, we write $(\cdot)\%n$ for the "modulo n" operation.

We need to define how to calculate with these extended matrices. Briefly, if we multiply a matrix with such entries with a vector, we simply substitute the values of the vector elements into the operation. For example,

$$\left[\begin{array}{cc} (\cdot)\%4 & 1 \\ (\cdot)/4 & 2 \end{array}\right]\left[\begin{array}{c} 2 \\ 3 \end{array}\right] = \left[\begin{array}{c} 2\%4+3 \\ 2/4+6 \end{array}\right] = \left[\begin{array}{c} 5 \\ 6 \end{array}\right] \quad (4)$$

On the other hand, if we multiply such a matrix by another matrix, we multiply the input or result of the operation by the appropriate integer. For example,

$$\left[\begin{array}{cc} 1 & 4 \end{array}\right] \times \left[\begin{array}{c} (\cdot)\%4 \\ (\cdot)/4 \end{array}\right] = [(\cdot)\%4 + 4*((\cdot)/4)] \quad (5)$$

This transformation maps an integer n to $n\%4 + 4*(n/4)$. It can easily be checked that $n\%4+4*(n/4) = n$ and hence this transformation can be replaced by the 1×1 identity matrix. Now on the index space of `A(0:3,0:7)`,

$$\left[\begin{array}{c} (\cdot)\%4 \\ (\cdot)/4 \end{array}\right] \times \left[\begin{array}{cc} 1 & 4 \end{array}\right] = \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right] \quad (6)$$

Hence these matrices are inverse to one another over the index space of `A`. For a more formal discussion of how to incorporate `div` and `mod` operations in a matrix framework, consult the full version of this paper.

4.4. Example

To illustrate this formulation, consider the program in figure 2 and the following transformation which maps the 2 dimensional array `A` to a 1 dimensional linearised form:

$$L = \left[\begin{array}{cc} 1 & 4 \end{array}\right] \text{ and } L^\dagger = \left[\begin{array}{c} (\cdot)\%4 \\ (\cdot)/4 \end{array}\right] \quad (7)$$

In the previous section we have shown that these matrices are local inverses. The array accesses to `A` are updated thus:

$$\left[\begin{array}{cc} 1 & 4 \end{array}\right] \times \left[\begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array}\right]\left[\begin{array}{c} j \\ i \end{array}\right] = \left[\begin{array}{cc} 4 & 1 \end{array}\right]\left[\begin{array}{c} j \\ i \end{array}\right] \quad (8)$$

$$\left[\begin{array}{cc} 1 & 4 \end{array}\right] \times \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right]\left[\begin{array}{c} j \\ i \end{array}\right] = \left[\begin{array}{cc} 1 & 4 \end{array}\right]\left[\begin{array}{c} j \\ i \end{array}\right] \quad (9)$$

i.e., `A(i,j)` $\mapsto$ `A(i+4*j)` and `A(j,i)` $\mapsto$ `A(j+4*i)`.

The new index space $\mathbf{A}'\mathcal{I}' \leq \mathbf{a}'$ is calculated as follows:

$$\mathbf{A}' = \begin{bmatrix} 1 & 4 & 0 & 0 \\ 0 & 0 & 1 & 4 \end{bmatrix} \left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \begin{bmatrix} (\cdot)\%4 \\ (\cdot)/4 \end{bmatrix} = \left[\begin{array}{c} -1 \\ \hline 1 \end{array}\right] \tag{10}$$

$$\mathcal{I}' = \begin{bmatrix} 1 & 4 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} j_1' \end{bmatrix} \tag{11}$$

$$\mathbf{a}' = \begin{bmatrix} 1 & 4 & 0 & 0 \\ 0 & 0 & 1 & 4 \end{bmatrix} \left[\begin{array}{c} 0 \\ 0 \\ \hline 3 \\ 7 \end{array}\right] = \left[\begin{array}{c} 0 \\ \hline 31 \end{array}\right] \tag{12}$$

giving

$$\left[\begin{array}{c} -1 \\ \hline 1 \end{array}\right] \begin{bmatrix} j_1' \end{bmatrix} \leq \left[\begin{array}{c} 0 \\ \hline 31 \end{array}\right] \tag{13}$$

i.e., `A(0:31)`. The updated array access and index space are shown in figure 3, row 1, column 2. The application of a similar loop transformation is shown in column 3.

4.5. Combining Loop and Data Transformations

Although our formulation allows rank modifying loop and data transformations to be applied independently, in practice they are often combined. For instance, the strip-mining data transformation gives rise to access functions containing `div` and `mod`, both of which will be prohibitively expensive. If, however, the surrounding loop were also strip-mined, the accesses would become simplified. For example, consider the code in figure 3, row 2, column 2, after the application of a rank-increasing data transformation. If a related *loop transformation* is subsequently applied, we have the program shown in row 2, column 4, where the accesses to `A` are now simplified. Thus, this framework allows a natural method to apply dimension changing transformations, frequently avoiding the use of `div` and `mod` without the need for special optimisations akin to strength reduction as described in [1].

5. Application : Handling Data Transformations in the Presence of Reshaped Arrays

This section is concerned with developing a technique to allow the application of data transformations to reshaped arrays without incurring excessive communication and synchronisation overhead. We give a compiler algorithm to generate the appropriate transformations and show experimental results.

5.1. Combining Loop and Data Transformations for Reshaped Arrays

This section describes a technique to apply data transformations to linearised arrays. This is followed by a section describing how rank modifying loop transformation can remove some of the introduced `div` and `mod` operators. This is then generalised to reshaped arrays.

5.1.1 Data Transformations on Linearised Arrays

Data transformations must be applied to every reference to the particular array throughout the program. Difficulties occur when linearised references exist. Let $\mathcal{I}_1$ be the index space of the array to be transformed and $\mathcal{I}_2 = L\mathcal{I}_1$ be the linearised space for some linearisation matrix L. We therefore have that $\mathcal{I}_1 = L^\dagger \mathcal{I}_2$.

If we wish to apply a non-singular data transformation $\mathcal{A}$ globally [12], this gives the new index domain $\mathcal{I}_1' = \mathcal{A}\mathcal{I}_1$. Since $\mathcal{I}_2' = L'\mathcal{I}_1'$ for some L',

$$\mathcal{I}_2' = L'\mathcal{I}_1' = L'\mathcal{A}\mathcal{I}_1 = L'\mathcal{A}L^\dagger\mathcal{I}_2 \tag{14}$$

Thus, when applying $\mathcal{A}$ to the index domain $\mathcal{I}_1$, we must apply $L'\mathcal{A}L^\dagger$ to the linearised domain $\mathcal{I}_2$. Now, given two references $\mathcal{U}_1$ and $\mathcal{U}_2$, where $\mathcal{U}_2$ is a linearised reference, then on applying $\mathcal{A}$ we have as usual $\mathcal{U}_1' = \mathcal{A}\mathcal{U}_1$. However, for the linearised access we have:

$$\mathcal{U}_2' = L'\mathcal{A}L^\dagger\mathcal{U}_2 \tag{15}$$

L and L' and their inverses are readily derived from the array bounds before and after applying $\mathcal{A}$.

5.1.2 Reducing Access Overhead for Linearised Arrays

Rank modifying transformations will, in general, introduce `mod` and `div` operations. We wish to remove these by introducing new iterators. The operators are in this case introduced the $L^\dagger$ matrix in equation (14). If we can apply a transformation such that this $L^\dagger$ is eliminated, then the corresponding comples accesses will be also be eliminated.

Let the loop transformation T be defined as follows:

$$T = \mathcal{U}^{-1}L^\dagger\mathcal{U} \quad \text{and} \quad T^{-1} = \mathcal{U}^{-1}L\mathcal{U} \tag{16}$$

Applying T to $\mathcal{U}_2'$ gives the new access matrix:

$$\mathcal{U}_2'T^{-1} = L'\mathcal{A}L^\dagger\mathcal{U}\mathcal{U}^{-1}L\mathcal{U} = L'\mathcal{A}\mathcal{U} \tag{17}$$

which is free from any rank-increasing matrices. Thus, by combining loop and data transformations within one framework, we can can readily restructure programs so as to partially undo the effect of previous transformations.

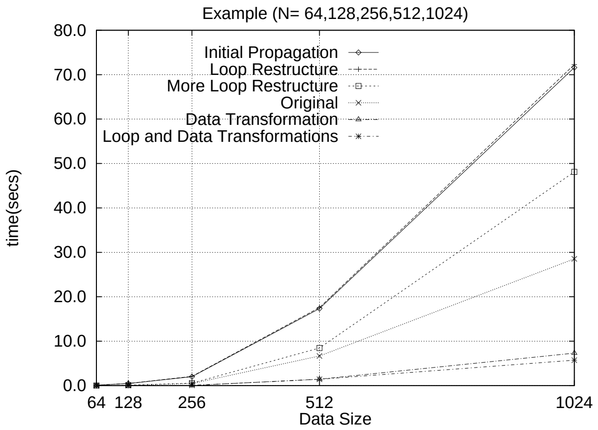


Figure 4. Example: $N = 64, \ldots, 1024$

5.1.3 Data Transformations for Reshaped Arrays

Application of data transformations for linearised accesses is relatively straightforward in the sense that it is easy to determine both L' and $L^\dagger$. Difficulties occur with reshaped arrays in that the shape of the array after application of a data transformation is not fixed, i.e., there are several legal new array layouts. A reshaped array access can be considered to be described by the following equation:

$$\mathcal{I}_2 = SL\mathcal{I}_1 \quad (18)$$

The reshaped domain $\mathcal{I}_2$ is considered to be constructed by first linearising $\mathcal{I}_1$ to a flat one-dimensional array which is then strip-mined to the appropriate dimension and size. The value of L is readily available given $\mathcal{I}_1$. So for any $\mathcal{I}_2$, the matrix S is easily derived. Note that $S \times L \neq 1$, otherwise no reshaping takes place. We therefore have $\mathcal{I}_1 = L^\dagger S^\dagger \mathcal{I}_2$. If we wished to apply a data transformation $\mathcal{A}$, this gives the new index domain $\mathcal{I}'_1 = \mathcal{A}\mathcal{I}_1$ and from equation (18) we have $\mathcal{I}'_2 = S'L'\mathcal{I}'_1$ and therefore

$$\mathcal{I}'_2 = S'L'\mathcal{I}'_1 = S'L'\mathcal{A}\mathcal{I}_1 = S'L'\mathcal{A}L^\dagger S^\dagger \mathcal{I}_2 \quad (19)$$

An access $\mathcal{U}_2$ to the reshaped array is transformed to $\mathcal{U}'_2$:

$$\mathcal{U}'_2 = S'L'\mathcal{A}L^\dagger S^\dagger \mathcal{U}_2 \quad (20)$$

We have S, L and thus $L^\dagger$. Again, L' is readily determined after applying $\mathcal{A}$ to $\mathcal{I}_1$. The difficulty occurs when deriving S' as there are no restrictions on its form except for legality. In other words the new dimensions of the reshaped array are not fixed after applying a data transformation on the original domain. To illustrate this, consider the original program in figure 1 box 1 and the reshaped access to array `A`, namely, `B` in box 2. If we choose $S' = S$ thereby preserving the original shape of the array, we have the program in figure 1, box 6. As is immediately apparent, though correct, this access function to `B` will be extremely costly.

1. Given the original reshaped array indices $\mathcal{I}_2$, determine the transformations L, S, $L^\dagger$ and $S^\dagger$.
2. Given the data layout transformation $\mathcal{A}$, update all reshaped array accesses such that $\mathcal{U} = \mathcal{A}L^\dagger S^\dagger \mathcal{U}$
3. Update the reshaped array indices such that $\mathcal{I}'_2 = \mathcal{A}L^\dagger S^\dagger \mathcal{I}_2$.
4. Update the array declaration $\mathbf{A}'\mathcal{I}'_2 \leq \mathbf{a}'$ accordingly.
5. Let $T = \mathcal{A}L^\dagger$.
6. For each loop nest containing reference to the array, apply the loop transformation T, if it is legal to do so.

Figure 5. Propagation Algorithm

5.1.4 Reducing Acess Overhead for Reshaped Arrays

In this subsection we examine two methods to reduce overhead due to global data transformations by applying a loop transformations to remove some of those rank-increasing transformations which introduce `mod`s. Let T be the following loop transformation

$$T = \mathcal{U}^{-1}L^\dagger S^\dagger \mathcal{U} \quad \text{and} \quad T^{-1} = \mathcal{U}^{-1}SL\mathcal{U} \quad (21)$$

Applying such a transformation would have the following affect on the accesses $\mathcal{U}'_2$ described in (20):

$$\mathcal{U}'_2 T^{-1} = S'L'\mathcal{A}L^\dagger S^\dagger \mathcal{U}T^{-1} = S'L'\mathcal{A}\mathcal{U} \quad (22)$$

If this transformation is applied to the code in box 6 in figure 1, we produce the program shown in box 9. There still remain `mod`s due to S'.

A possibly more straightforward approach would be to select the reshaping matrix S' to be $S' = L'^\dagger$. This has the effect that arrays are remapped onto the same shape as the original array when applying a propagated data transformation $\mathcal{A}$. Thus equation (19) simplifies to:

$$\mathcal{I}'_2 = L'^\dagger L'\mathcal{A}L^\dagger S^\dagger \mathcal{I}_2 = \mathcal{A}L^\dagger S^\dagger \mathcal{I}_2 \quad (23)$$

If we apply this data transformation to the code in box 6 of figure 1, we arrive at the code in box 12. Finally, a simple method to improve performance is to examine the array access matrix and then strip-mine those iterators which are arguments of `div` or `mod`, and reorder the iterators to give the code in box 13 with the stride-1 access shown in box 14. In other words, apply a loop transformation $T = \mathcal{A}L^\dagger$ and update the loop bounds and array accesses accordingly.

5.2. Algorithm

Given the techniques developed in section 5.1, it is relatively straightforward to incorporate then into a compiler phase order. Consider the algorithm in figure 5. Within our optimising compiler, this algorithm is applied after the global data partitioning/alignment has been chosen and after barrier synchronisation placement has been determined, but before loop optimisations and code generation. Once our compiler has determined the global data layout, it must consider reshaped arrays (step 1). The application of the reshaped data transformation (step 2) is followed by the update of array declarations (steps 3 and 4) before a loop transformation is constructed (step 5) to remove any remaining `mods` etc (step 6). As step 6 may reorder the loop nest, its legality must be checked before application.

5.3. Results

To show the use of the analysis developed in this section, we ran each of the program versions shown in figure 1 on the SGi Challenge for varying data sizes. Figure 4 shows their relative performance compared to the original program. The basic data propagation scheme is more than twice as poor as the original. Although subsequent loop transformations do not improve performance, they still do match the performance of the original. When both loop and data transformations are applied we finally have an improvement over the original. For $N = 1024$ this improvement is by over a factor of 4.

For a more realistic evaluation, we applied the algorithm shown in figure 5 to the SPECfp92 benchmark `vpentatest`. This program was selected as it contains a subroutine call where two of the array arguments are reshaped. Both programs were parallelised by our compiler [3] and the new algorithm was compared to the existing technique of remapping array data at run time. The two versions were compared for data sizes $n = 128$, 256, 512 and 1024 and executed on an SGi Origin 2000 with optimisation level -O3. The results are shown in figures 6, 7, 8 and 9, where the x-axis corresponds to the number of processors and the y-axis to 1/time in seconds. As can be seen, the code generated by the algorithm (labelled "New") is between a factor of 2 and 4 times faster than the standard scheme (labelled "Remap"). For the smallest data size, however, neither implementation gives any speed-up. This is due to the compiler deciding to partition on the first dimension, leading to false sharing as the number of processors increases.

6. Related Work

There has been some recent work in integrating loop and data transformations. Although loop transformation have been generalised from a unimodular [2] to a non-singular framework [11] and have been extended to statement-wise transformations [10], the data transformations considered

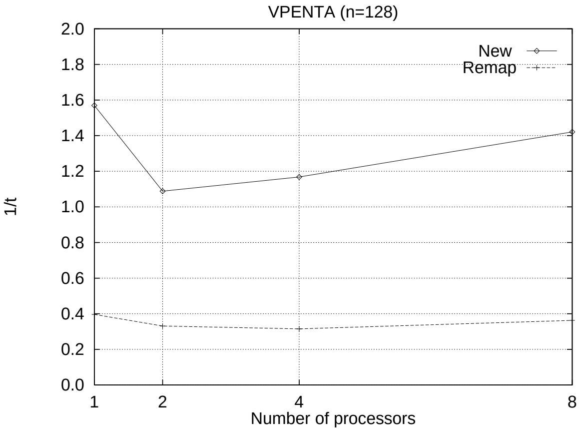


Figure 6. VPENTA: N = 128

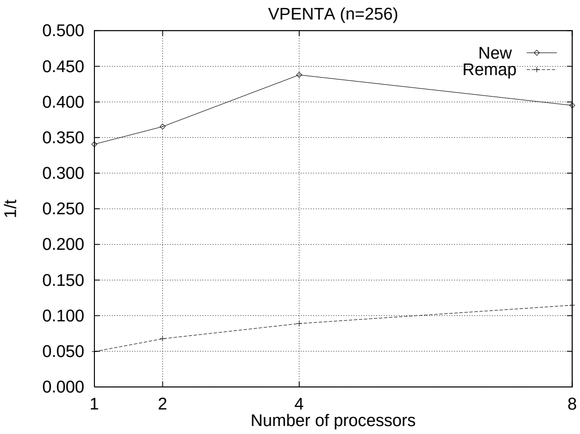


Figure 7. VPENTA: N = 256

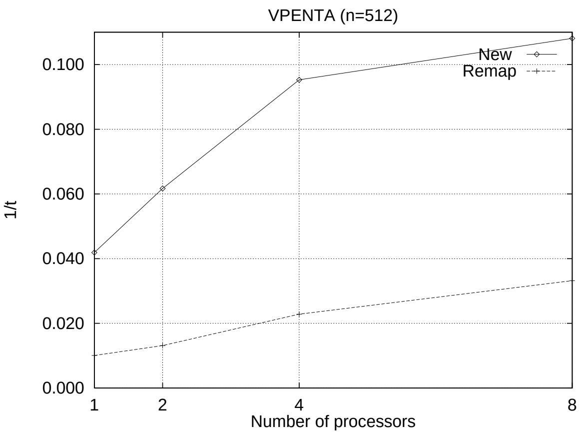


Figure 8. VPENTA: N = 512

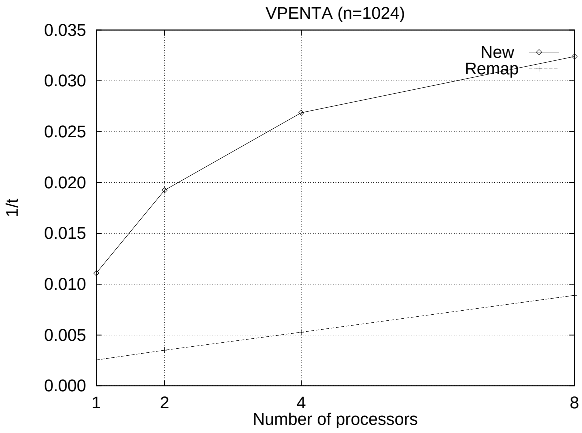


Figure 9. VPENTA: N = 1024

are more restricted. For instance, [4] only considers linearisation while [1] allows just permutation and strip-mining. In [1], loop based parallelisation is followed by array layout modification to enhance spatial locality. In [4], a technique combining restricted data transformations with unimodular loop transformations is developed to improve memory access patterns. This representation is subsequently used in [7] to develop an alternative algorithm to improve memory access behaviour. In [12], we extended the class of data-transformations to the general non-singular but were unable to incorporate linearisation or partitioning. In [5], the general case of array aliasing, particularly across array boundaries, is considered and preliminary techniques to recover the structure of linearised arrays are developed. They also develop ad hoc techniques to recover loop structure after data restructuring but cannot, at present, handle the application of data transformations, such as data partitioning, across aliased arrays. The techniques developed in this paper, however, allow the static application of data transformations, including array partitioning across reshaped arrays, providing the necessary results for [5].

7. Conclusion

In this paper, we have presented a new algebraic framework that allows the integration of loop and data transformations. This has enabled existing transformations to be described in a unifying manner and has also provided the basis for new program optimisations. In particular, we have developed techniques which allow the application of optimising data transformations to reshaped arrays without incurring excessively expensive code. Future work will consider integrating this with complete inter-procedural analysis. Future work will further investigate the mathematical properties of the transformation representation used in this paper and, in particular, develop formal validity tests and investigate further optimisation algorithms.

References

[1] J.M. Anderson S.P. Amarasinghe and M.S. Lam. **Data and Computation Transformations for Multiprocessors**, Proc. PPoPP, 1995.

[2] U. Banerjee. **Loop Transformations for Restructuring Compilers**, Kluwer Academic Publishers, 1993.

[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] M. Cierniak and W. Li. **Unifying Data and Control Transformations for Distributed Shared-Memory Machines**, Proc. PLDI, 1995.

[5] M. Cierniak and W. Li. **Validity of Interprocedural Data Remapping**, Tech Rep 642, University of Rochester, 1996.

[6] T.E.Jeremiassen and S.J. Eggers, **Reducing False Sharing on Shared Memory Multiprocessors through Compile-Time Data Transformations**, Proc. PPoPP, 1995.

[7] M. Kandemir, J. Ramanujam and A. Choudhary, **A Compiler Algorithm for Optimizing Locality in Loop Nests**, Proc. ICS, 1997.

[8] K. Kennedy and U. Kremer. **Automatic Data Layout for High Performance Fortran**, Proc. Supercomputing, 1995.

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

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

[11] W. Li and K. Pingali. **A Singular Loop Transformation Framework Based on Non-singular Matrices**, Int'l J. of Parallel Programming, **22**(2), pp. 183–205, 1994.

[12] M.F.P. O'Boyle and P.M.W. Knijnenburg, **Non-Singular Data Transformations: Definition, Validity and Applications**, Proc. ICS, 1997.

[13] M.E. Wolf and M. Lam. **A Loop Transformation Theory and An Algorithm to Maximise Parallelism**, ACM Transactions on Parallel and Distributed Systems 2(4), October 1991.