Integrating Loop and Data Transformations for Global Optimisation

M.F.P. O'Boyle

Institute for Computing Systems Architecture
Division of Informatics
The University of Edinburgh
Edinburgh EH9 3JZ
United Kingdom

E-mail: `mob@dcs.ed.ac.uk`

and

P.M.W. Knijnenburg

Leiden Institute of Advanced Computer Science
Leiden University
Niels Bohrweg 1
2333 CA Leiden
the Netherlands

E-mail: `peterk@liacs.nl`

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, poor spatial locality and expensive array subscripts that may have been introduced can be eliminated. A specific optimisation algorithm is derived and applied to well-known benchmarks, where it is shown to give a significant improvement in execution time over existing approaches.

Key Words: Compiler Optimisation, Unified Program Transformations, Rank-Modifying Transformations, Reshaped Arrays, Auto-parallelisation.

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 [25] 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 [14], traditionally developed for distributed memory compilation but also used for distributed shared memory [1, 10]. 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 storage elements. Although there has been much work in inter-procedural analysis, 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 [7], 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.

This paper focuses on these two problems of data parallelisation, namely, reshaped arrays and the adverse effect on loop nests. 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, 6, 11, 23] has focused on combining the loop based parallelisation with data layout approaches [14] 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 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. This may be incorporated with inter-procedural analysis [1] to allow data transformations across procedure boundaries. Applying global data transformations to reshaped arrays can, however, produce complex and inefficient code but, by integrating loop and data transformations, we may 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. Sections 3 and 4 describe the extended matrix formulation and the notation used within this paper respectively. This is followed by section 5 which describes the form and properties of our novel rank modifying transformations. Sections 6 and 7 develop techniques for efficient data layout propagation which is followed in section 8 by a compiler algorithm and experimental results showing the significant improvement of such a scheme. Section 9 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: application of data transformations to reshaped arrays and the impact of global data transformations [2] on local loop structure. To illustrate one of the applications, consider the program fragment in figure 1, position 1. In Fortran arrays are stored column-wise and therefore the reference to array `A` has a non stride-1 access pattern. This is shown in the diagram in position 3. The program fragment in position 2 contains references to `B` which are reshaped references to `A`. Here, however, there is perfect stride-1 access to array `B` as shown in position 4. One way to improve the access to array `A` is to interchange the two loops. Loop permutation of the code in position 1 is not possible, however, due to the data dependence $[1,-1]^T$and thus a compiler may wish to apply a data transformation, described in [23], to ensure stride-1 access. The necessary data transformation is a simple permutation matrix which, when applied to array `A`, gives the code in position 5 with the corresponding stride-1 access pattern in position 7. Such an approach has the benefit of always being legal, though its effect must be propagated to all accesses, including reshaped ones. Existing approaches [7] do not propagate information when there is reshaping and prefer either to insert a copy or prevent the data restructuring. Using the techniques described in section 6, we can apply the data transformation to the reshaped array giving the code in position 6 and the access pattern shown in position 8. In general there will be conflict if there are two or more caller sites to the same subroutine with different shapes. Although the necessary inter-procedural analysis is beyond the scope of this paper, one obvious approach would be to clone the callee subroutine if there are conflicting caller shapes [15]. Although correct, it is immediately apparent that

[1]beyond the scope of this paper

[2]in this paper, we use the term data transformation to denote any transformation that changes the declaration of an array and its subscripts when referenced. This encompasses transformations such as global index reordering and linearisation

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)**

FIG. 1. Effect of Propagated Data Transformation

such a code structure (position 6) and access pattern (position 8) will be expensive, outweighing any benefit to the improved stride access to `A`. Even after application of the strength-reduction techniques for `mod` operations described in [1], calculation of the access function of `B` will be prohibitive. Furthermore, this will not improve the pathological "leapfrog" access to `B`.

The table in figure 2 shows different attempts to improve the loop structure and data access pattern of the code in position 6, figure 1. Using a combination of strip-mining and linearisation, the loop can be transformed to that shown in position 9, figure 2. The access pattern, shown in position 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 position 10, where all `mod` and `div` operators have been removed. This, however, does not affect the data access

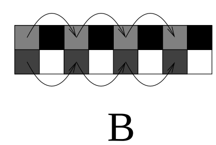
Loop Restr. (9)

```
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) =
  B(mod(j,2),2*i+j/2)+1
 Enddo
Enddo
```

More Loop Restr. (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) =
    B(j2,2*i+j1)+1
  Enddo
 Enddo
Enddo
```

Access Pattern(11)



Data Restr. (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)) =
  B(j/2,i+2*mod(j,2))+1
 Enddo
Enddo
```

Loop + Data Restr. (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) =
    B(j2,i+2*j1)+1
   Enddo
  Enddo
 Enddo
Enddo
```

Access Pattern(14)

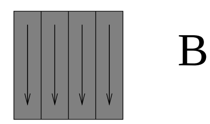


FIG. 2. Transformations

pattern shown in position 11. If instead a data transformation is applied to `B`, we have the code shown in position 12. Finally, if both loop and data transformations are combined, we obtain the code in position 13 which has stride-1 access shown in position 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 7 we develop automatic techniques which select the appropriate transformations and show the impact of such restructuring on actual performance.

3. EXTENDED MATRICES

This section presents the extended matrix formulation used to describe the rank-modifying transformations used in propagating data layouts in the presence of reshaped arrays 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 and $n \times (\cdot)$ for "multiply by n".

The properties of these additional terms, on entries of a singleton matrix, are defined as follows:

DEFINITION 3.1. For a and $n \in \mathbf{Z}$, we define

$$\begin{aligned} [n \times (\cdot)]\,[a] &= [n \times a] \\ [(\cdot)/]\,[a] &= [n/a] \\ [(\cdot)\%n]\,[a] &= [n \bmod a] \end{aligned}$$

This can readily be generalised to *extended matrices*:

DEFINITION 3.2. We define an $n \times m$ *extended matrix* as an $n \times m$ array of functions $f_{ij} : \mathbf{Z} \to \mathbf{Z}$ for $1 \leq i \leq n$ and $1 \leq j \leq m$ where f_{ij} is restricted to functions defined in Definition 3.1.

Function application rather than multiplication is needed when considering the composition of extended matrices.

DEFINITION 3.3. The application of an $n \times m$ extended matrix A to a vector $b \in \mathbf{Z}^m$ is a vector $c = Ab \in \mathbf{Z}^n$ defined as follows. For all $1 \leq k \leq n$,

$$c_k = \sum_{i=1}^{m} f_{ki} b_i \tag{1}$$

Ordinary matrices can easily be incorporated in this new framework. Consider, initially, a 2×2 standard integer matrix A and a vector b

$$A = \begin{bmatrix} a_1 & a_2 \\ a_3 & a4 \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Then the application $A(b)$ is defined as

$$A(b) = \begin{bmatrix} a_1 \times b_1 + a_3 \times b_2 \\ a_3 \times b_1 + a_4 \times b_4 \end{bmatrix} \tag{2}$$

where both multiplication and addition of (natural) numbers are used for this operation.

Now, the matrix A can be rewritten as an extended matrix A' where

$$A' = \begin{bmatrix} a_1 \times (\cdot) & a_2 \times (\cdot) \\ a_3 \times (\cdot) & a_4 \times (\cdot) \end{bmatrix}$$

and thus

$$A'(b) = \begin{bmatrix} a_1 \times (\cdot) & a_2 \times (\cdot) \\ a_3 \times (\cdot) & a_4 \times (\cdot) \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1 \times (b_1) + a_2 \times (b_2) \\ a_3 \times (b_1) + a_4 \times (b_2) \end{bmatrix} = A(b) \tag{3}$$

It is easy to see that for all vectors $b \in \mathbf{Z}^2$, $Ab = A'b$ with the above definition of A' and matrix-vector multiplication. In this sense, the new framework is an extension of ordinary matrices.

From now on, for the sake of readability, we will write matrix entries involving multiplication such as $k \times (\cdot)$ for some integer k, simply as k. In this way, the notation of ordinary matrices, with only integer entries, is exactly the same as in the traditional notation and so is their action on integer vectors. With modulo and division entries, the vector elements are simply substituted. For example:

$$\begin{bmatrix} (\cdot)\%4 & 1 \\ (\cdot)/4 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} (2\%4)+3 \\ (2/4)+6 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix} \tag{4}$$

3.1. Combining Extended Matrices

Frequently we wish to combine several extended matrices. Standard integer matrices are associative and sequences may be reduced to a single integer matrix by multiplication. In this section we define the composition of two or more extended matrices and show that in general they may not be reduced to a single matrix.

Consider two 2×2 extended matrices A and B.

$$A = \begin{bmatrix} f_1 & f_2 \\ f_3 & f_4 \end{bmatrix} \qquad B = \begin{bmatrix} g_1 & g_2 \\ g_3 & g_4 \end{bmatrix}$$

and an arbitrary vector $a \in \mathbf{Z}^2$. Then $A(B(a))$ can be written as

$$A(B(a)) = \begin{bmatrix} f_1(g_1(a_1)+g_2(a_2)) + f_2(g_3(a_1)+g_4(a_2)) \\ f_3(g_1(a_1)+g_2(a_2)) + f_4(g_3(a_1)+g_4(a_2)) \end{bmatrix}$$

We say that a function f is *linear* if $f(a+b) = f(a) + f(b)$. For example, multiplication is linear: $n \times (a+b) = n \times a + n \times b$. An extended matrix A is called linear if all its elements are linear functions. In the example above, if A and B are linear, then their composition $A \circ B$ can be written as

$$A \circ B = \begin{bmatrix} f_1 \circ g_1 + f_1 \circ g_2 & f_1 \circ g_2 + f_2 \circ g_4 \\ f_3 \circ g_1 + f_4 \circ g_3 & f_3 \circ g_2 + f_4 \circ g_4 \end{bmatrix}$$

This is the extension of ordinary matrix multiplication.

THEOREM 3.1. *In the case when the extended matrices A and B are linear, then their composition $A \circ B$ can be written as one single extended matrix C.*

Proof. Above we have shown this to be true for the 2×2 case. The general case for $n \times n$ matrices follows by induction on n ■

Thus the new framework truly extends the traditional framework of matrices. Extended matrix multiplication has exactly the same form as as ordinary matrix multiplication, but with function composition instead of multiplication of elements, when we restrict attention to linear functions.

However, neither integer division nor modulo is linear and thus we can not in general simplify sequences of extended matrices into one single matrix, except in special circumstances (based on inverses) described in section 5. As function composition is associative so is extended matrix composition but, like standard matrix composition, it is not commutative. If we assume that all the operations in Definition 3.1 plus addition have unit complexity, then matrix composition has the same complexity as standard matrix multiplication namely $O(n^3)$.

4. NOTATION

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

4.1. 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*

$$\mathbf{B}J \leq \mathbf{b} \tag{5}$$

where $\mathbf{B}$ is a $(\ell \times m)$ integer matrix and $\mathbf{b}$ an $(\ell \times 1)$ vector, for some ℓ. The integer values taken on by J define the *iteration space* of the iterators.

4.2. Index Spaces

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

$$\mathcal{I} = [i_1, i_2, \ldots, i_N]^T$$

where N 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:

$$\mathbf{A}\mathcal{I} \leq \mathbf{a} \tag{6}$$

where $\mathbf{A}$ is a $(\ell \times N)$ integer matrix and $\mathbf{a}$ an $(\ell \times 1)$ vector, for some ℓ. The integer values taken on by $\mathcal{I}$ define the *index space* of the indices.

4.3. 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 domain. If J is the iteration vector, we assume in this paper that these subscripts can be written in the following form:

$$\mathcal{U}J + u \tag{7}$$

where $\mathcal{U}$ is a $N \times m$ matrix and u is a $N \times 1$ vector. All access vectors and loop and array bounds are assumed to start from 0 throughout this paper, for ease of presentation. The transformations can be trivially extended to encompass non-zero offsets.

5. RANK MODIFYING TRANSFORMATIONS

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

5.1. Form of Transformations

In this paper, we restrict attention to generalised strip-mining and linearisation based on extended matrices described in section 3. The simplest linearisation matrix $\mathbf{L}$ is a 1×2 matrix which maps a 2 dimension iteration or index vector to a 1 dimensional vector.

$$\mathbf{L} = \left[\begin{array}{cc} 1 & n_1 \end{array}\right] \tag{8}$$

where n_1 is a constant corresponding to the size of the first index/iterator dimension.

For a linear algebraic transformation representation to be useful, it must have an inverse [23]. Rank deficient matrices such as $\mathbf{L}$ are singular and do not have a true inverse, but it is possible to construct a *pseudo* inverse $\mathbf{L}^\dagger$ that is valid on a sub-space D^2 of $\mathbf{Z}^2$. That is, for each $a \in D$, $\mathbf{L}^\dagger(\mathbf{L}(a)) = a$. This statement need not hold for elements in $\mathbf{Z}^2$ outside D^2. However, in practice, this sub-space D corresponds to the index domain of an array, or the iterator domain of a loop, to which $\mathbf{L}$ is to be applied. Now, although $\mathbf{L}$ is a simple integer matrix, its pseudo inverse $\mathbf{L}^\dagger$ includes `mod`s and `div`s, hence our need for the extended matrix representation.

$$\mathbf{L}^\dagger = \left[\begin{array}{c} (\cdot)\%n_1 \\ (\cdot)/n_1 \end{array}\right] \tag{9}$$

Next, $\mathbf{S}$ is the simplest strip-mining 2×1 matrix which maps a 1-dimensional iteration or index vector to a 2 dimensional vector.

$$\mathbf{S} = \left[\begin{array}{c} (\cdot)\%n_1 \\ (\cdot)/n_1 \end{array}\right] \tag{10}$$

where n_1 is a constant corresponding to the strip-mine factor and has the following inverse

$$\mathbf{S}^\dagger = \left[\begin{array}{cc} 1 & n_1 \end{array}\right] \tag{11}$$

THEOREM 5.1. *$\mathbf{L}$ has an inverse $\mathbf{L}^\dagger$ and $\mathbf{S}$ has an inverse $\mathbf{S}^\dagger$ such that $\mathbf{L}^\dagger(\mathbf{L}(a)) = a$ and $\mathbf{S}^\dagger(\mathbf{S}(a)) = a$ where a corresponds to an iteration/index vector.*

Proof. See appendix ■

As is apparent from the above equations, $\mathbf{L}$ and $\mathbf{S}$ are inverses to each other which follows from the intuition that strip-mining a loop then linearising it returns it to its original form. Thus $\mathbf{L}$ and $\mathbf{S}^\dagger$ have the same form, as do $\mathbf{S}$ and $\mathbf{L}^\dagger$.

In general we may wish to linearise the kth element in an N-dimensional iteration/index vector. This can be achieved by embedding $\mathbf{L}$ within a matrix to give the general linearisation matrix L:

$$L = \begin{bmatrix} I_{k-1} & O & O \\ O & \mathbf{L} & O \\ O & O & I_{N-k} \end{bmatrix} \tag{12}$$

where I_m is the m dimensional identity matrix and $\mathbf{L} = \begin{bmatrix} 1 & n_k \end{bmatrix}$ where n_k is the size of the kth dimension.

THEOREM 5.2. *L has an inverse $L^{\dagger}$.*

Proof. See appendix ■

$\mathbf{S}$ can be similarly embedded to give a generalised strip-mining matrix S. Linearisation or strip-mining of more than one dimension can be achieved by the appropriate sequence of L or S matrices, respectively.

5.2. 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. This means that every access to this transformed array must be updated as a data transformation is in effect an array element-wise renaming procedure [23]. Rank-decreasing data transformations, such as array linearisation, are frequently used in FORTRAN programs, while rank-increasing data transformations are used when partitioning data for multi-processor parallel execution.

A $(k \times N)$ linearisation matrix L is a transformation which maps an N dimensional index vector $\mathcal{I}$ to a new k dimensional vector $\mathcal{I}'$.

$$\mathcal{I}' = L\mathcal{I} \tag{13}$$

Each array access $\mathcal{U}$ must be globally updated such that

$$\mathcal{U}' = L\mathcal{U} \tag{14}$$

Data transformations are therefore *left-hand* transformations when applied to array access functions. The new bounds of the new iteration space must also be determined. They are of the form:

$$\mathbf{A}'\mathcal{I}' \leq \mathbf{a}' \tag{15}$$

where

$$\mathbf{A}' = X\mathbf{A}L^{\dagger} \quad \text{and} \quad \mathbf{a}' = X\mathbf{a} \tag{16}$$

and

$$X = \begin{bmatrix} L & O \\ O & L \end{bmatrix} \tag{17}$$

In equation (16), $L^\dagger$ is the pseudo inverse to L on the index space of `A`.

For rank decreasing transformations, the only additional condition to be added is that $k = N-1$, i.e., the new index space is smaller than the original. The equations for applying a rank-increasing transformation, such as strip-mining, have precisely the same form where $k = N+1$ and in equations 13 to 17, L is replaced by S.

5.3. Loop Transformations

Loop transformations are applied to the iterators in a loop nest and to all array accesses within a loop nest and are thus *local* in nature. Rank-decreasing loop transformations, such as loop collapsing, are used to reduce loop overhead while rank-increasing ones, such as tiling, are frequently used to exploit locality.

A $(k \times m)$ linearisation matrix L is a transformation which maps an m dimensional index vector J to a new k dimensional vector J'.

$$J' = LJ \tag{18}$$

Each access $\mathcal{U}$ within the loop nest must be updated such that

$$\mathcal{U}' = \mathcal{U}L^\dagger \tag{19}$$

Thus, loop transformations are *right-hand* acting transformations when applied to array accesses.

The new bounds of the new iteration space must be determined and are of the form:

$$\mathbf{B}'J' \leq \mathbf{b}' \tag{20}$$

where

$$\mathbf{B}' = X\mathbf{B}L^\dagger \quad \text{and} \quad \mathbf{b}' = X\mathbf{b} \tag{21}$$

and

$$X = \begin{bmatrix} L & O \\ O & L \end{bmatrix} \tag{22}$$

Once again, the only additional condition to be added is that $k = N-1$ for rank-decreasing transformations. In the case of rank increasing matrices $k = N+1$ and in equations 18 to 22, L is replaced by S

5.4. Legality

Just as linearisation and strip-mining are always legal program transformations [16] so are the application of the rank-modifying transformations L and S.

```
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
```

FIG. 3. Example loop

THEOREM 5.3. *Rank-modifying transformations are always legal loop transformations*

Proof. See appendix ■

THEOREM 5.4. *Rank-modifying transformations are always legal data transformations*

Proof. See appendix ■

5.5. Example

To illustrate this rank-modifying formulation, consider the program in figure 3 and the following transformation which maps the 2 dimensional array `A` to a 1 dimensional linearised form:

$$L = \begin{bmatrix} 1 & 4 \end{bmatrix} \quad \text{and} \quad L^\dagger = \begin{bmatrix} (\cdot)\%4 \\ (\cdot)/4 \end{bmatrix} \tag{23}$$

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

$$\begin{bmatrix} 1 & 4 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} = \begin{bmatrix} 4 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} \tag{24}$$

$$\begin{bmatrix} 1 & 4 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} = \begin{bmatrix} 1 & 4 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} \tag{25}$$

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:

$$\mathbf{A}' = \begin{bmatrix} 1 & 4 & 0 & 0 \\ 0 & 0 & 1 & 4 \end{bmatrix} \times \left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \times \begin{bmatrix} (\cdot)\%4 \\ (\cdot)/4 \end{bmatrix} = \left[\begin{array}{c} -1 \\ \hline 1 \end{array}\right] \tag{26}$$

L, S	Data	Loop	Combined

$\left[\begin{array}{cc} 1 & 4 \end{array}\right]$	`Real A(0:31)`
	`Real 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)`
		`Real 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)`
			`Real 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)`
	`Real 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)`
		`Real 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)`
			`Real 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`

FIG. 4. Loop and Data Transformations

$$\mathcal{I}' = \left[\begin{array}{cc} 1 & 4 \end{array}\right] \left[\begin{array}{c} i_1 \\ i_2 \end{array}\right] = \left[\begin{array}{c} i_1' \end{array}\right] \tag{27}$$

$$\mathbf{a}' = \left[\begin{array}{cccc} 1 & 4 & 0 & 0 \\ 0 & 0 & 1 & 4 \end{array}\right] \left[\begin{array}{c} 0 \\ 0 \\ \hline 3 \\ 7 \end{array}\right] = \left[\begin{array}{c} 0 \\ \hline 31 \end{array}\right] \tag{28}$$

giving

$$\left[\begin{array}{r} -1 \\ \hline 1 \end{array}\right] \left[\begin{array}{c} i_1' \end{array}\right] \leq \left[\begin{array}{c} 0 \\ \hline 31 \end{array}\right] \tag{29}$$

i.e., `A(0:31)`. The updated array access and index space are shown in figure 4, row 1, column 2. The application of a similar loop transformation is shown in column 3.

5.6. 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 4, 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.

Now, as was stated in section 3, we cannot in general reduce a sequence of extended matrices to a single matrix. If we apply n rank-modifying transformations, we potentially have $O(n^2)$ terms containing `mod` and `div`. However, due to the construction of rank-modifying transformations, we can always reduce a transformation followed by its inverse to the identity matrix as shown in theorem 5.1. 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].

6. HANDLING DATA TRANSFORMATIONS IN THE PRESENCE OF LINEARISED ARRAYS

The previous sections have described the form of rank-modifying transformations. This section illustrates the usefulness of such a framework by combining rank-modifying transformations with more standard non-singular ones. It develops a technique to allow the application of data transformations to linearised arrays without incurring excessive communication and synchronisation overhead.

The first subsection 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.

6.1. Data Transformations on Linearised Arrays

Data transformations are left-hand transformations that must be applied to every reference to the particular array throughout the program. Difficulties occur when references to a linearised array access exist. Let $\mathcal{I}_1$ be the index domain of the array to be transformed and $\mathcal{I}_2$ be the the linearised domain:

$$\mathcal{I}_2 = L\mathcal{I}_1 \tag{30}$$

where L is the linearising transformation. We therefore have:

$$\mathcal{I}_1 = L^\dagger\mathcal{I}_2 \tag{31}$$

If we wish to apply a non-singular data transformation $\mathcal{A}$ globally [23], this gives the new index domain $\mathcal{I}_1'$:

$$\mathcal{I}_1' = \mathcal{A}\mathcal{I}_1 \tag{32}$$

and analogous to equation (30) we have

$$\mathcal{I}_2' = L'\mathcal{I}_1' \tag{33}$$

where L' is the appropriate linearisation and therefore

$$\mathcal{I}_2' = L'\mathcal{I}_1' = L'\mathcal{A}\mathcal{I}_1 = L'\mathcal{A}L^\dagger\mathcal{I}_2 \tag{34}$$

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

$$\mathcal{U}'_2 = L'\mathcal{A}L^\dagger\mathcal{U}_2 \tag{35}$$

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

To illustrate this, consider the example in figure 5, column 1. Let

$$\mathcal{A} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{36}$$

i.e., a simple data skew which is applied to the array `A`. The access to array `A` in the first loop is readily found:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} \tag{37}$$

i.e., `A(i,i+j)` which is shown in figure 5, column 2. To update the equivalenced access to array `B`, we need to determine L' and $L^\dagger$. Examining the original array bounds we have:

$$L = \begin{bmatrix} 1 & 2 \end{bmatrix} \quad \text{and} \quad L^\dagger = \begin{bmatrix} (\cdot)\%2 & (\cdot)/2 \end{bmatrix}^T \tag{38}$$

Applying $\mathcal{A}$ gives the new array bounds `A(0:1,0:4)`. Hence

$$L' = \begin{bmatrix} 1 & 2 \end{bmatrix} \tag{39}$$

If we apply $L'\mathcal{A}L^\dagger$ to the linearised access to `B` in the first loop we get:

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} (\cdot)\%2 \\ (\cdot)/2 \end{bmatrix} \times [2,1] \times \begin{bmatrix} j \\ i \end{bmatrix} = [2*j, 3*i] \tag{40}$$

i.e., `B(3*i + 2*j)` which is shown in figure 5 column 2. Repeating the procedure for the second loop we have:

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} (\cdot)\%2 \\ (\cdot)/2 \end{bmatrix} \times [1] \times [j] = [3*(j\%2) + 2*(j/2)] \tag{41}$$

i.e., `A(3*mod(j,2) + 2*(j/2))` Thus, our formulation allows systematic application of data transformations such as data alignment even in the presence of linearised array accesses.

6.2. Reducing Access Overhead for Linearised Arrays

Although correct, the program in figure 5, column 2 is far from ideal due to the `mod` and `div` operators. We wish to remove these by introducing two new iterator variables corresponding to `mod(j,2)` and `j/2` which is achieved by strip-mining.

Original	Data Transformation	Loop Transformation

`Real A(0:1,0:3)`
`Real B(0:7)`
`Equivalence(A,B)`
`.`
`.`
`Do j = 0,3`
` Do i = 0,1`
`    A(i,j ) =`
`      B(i+2*j)`
`    + C(i,j)`
` Enddo`
`Enddo`
`Do j = 0,7`
`  B(j) = j*3`
`Enddo`
	`Real A(0:1,0:4)`
	`Real B(0:9)`
	`Equivalence(A,B)`
	`.`
	`.`
	`Do j = 0,3`
	` Do i = 0,1`
	`   A(i,i+j) =`
	`     B(3*i+2*j)`
	`   + C(i,j)`
	` Enddo`
	`Enddo`
	`Do j = 0,7`
	`  B(3*mod(j,2)+ 2*(j/2))`
	`  = 9*mod(j,2)+ 6*(j/2)`
	`Enddo`
		`Real A(0:1,0:4)`
		`Real B(0:9)`
		`Equivalence(A,B)`
		`.`
		`.`
		`Do j = 0,3`
		` Do i = 0,1`
		`  A(i,i+j) =`
		`    B(3*i+2*j)`
		`  + C(i,j)`
		` Enddo`
		`Enddo`
		`Do j = 0,3`
		` Do i = 0,1`
		`  B(3*i + 2*j) =`
		`    9*i+6*j`
		` Enddo`
		`Enddo`

FIG. 5. Linearised Aliasing

The `mod` and `div` operators are introduced by rank-increasing transformation such as S or, in this case, the $L^{\dagger}$ matrix in equation (34). If we can apply a transformation such that this is eliminated, then the corresponding `mod(j,2)` and `j/2` will be eliminated.

Let T be defined as follows:

$$T = \mathcal{U}_2^{-1} L^{\dagger} \mathcal{U}_2 \tag{42}$$

and hence

$$T^{-1} = \mathcal{U}_2^{-1} L \mathcal{U}_2 \tag{43}$$

Applying this transformation to the loop and array accesses, using equation (35) gives the new access matrix:

$$\mathcal{U}_2' T^{-1} = L' \mathcal{A} L^{\dagger} \mathcal{U}_2 \mathcal{U}_2^{-1} L \mathcal{U}_2 = L' \mathcal{A} \mathcal{U}_2 \tag{44}$$

which is free from any rank-increasing matrices. Applying such a transformation to the second loop gives the code in column 3 of figure 5. 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 transformation applications.

7. HANDLING DATA TRANSFORMATIONS IN THE PRESENCE OF RESHAPED ARRAYS

This section extends the results of the previous section by developing a technique to allow the application of data transformations to reshaped arrays without incurring excessive communication and synchronisation overhead. However, this is at the cost of complex subscripts and poor strides access and we therefore examine loop transformations to reduce introduced overheads due to global data transformations.

7.1. 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:

$$\mathcal{I}_2 = SL\mathcal{I}_1 \tag{45}$$

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 $\mathcal{I}_2$, S is easily derived. For instance, if an array is declared of size `A(0:7,0:3)` in the main program but reshaped to size `A(0:1,0:15)` in a subroutine we have:

$$L = [1, 8] \text{ and } S = \begin{bmatrix} (\cdot)\%2 \\ (\cdot)/2 \end{bmatrix} \tag{46}$$

Note that $S \times L \neq I$ otherwise no reshaping takes place. We therefore have

$$\mathcal{I}_1 = L^\dagger S^\dagger \mathcal{I}_2 \tag{47}$$

If we wish to apply a data transformation $\mathcal{A}$, this gives the new index domain $\mathcal{I}_1'$

$$\mathcal{I}_1' = \mathcal{A}\mathcal{I}_1 \tag{48}$$

and analogous to equation (45) we have

$$\mathcal{I}_2' = S'L'\mathcal{I}_1' \tag{49}$$

for some S' and L'. Therefore

$$\mathcal{I}_2' = S'L'\mathcal{I}_1' = S'L'\mathcal{A}\mathcal{I}_1 = S'L'\mathcal{A}L^\dagger S^\dagger \mathcal{I}_2 \tag{50}$$

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

$$\mathcal{U}_2' = S'L'\mathcal{A}L^\dagger S^\dagger \mathcal{U}_2 \tag{51}$$

We have S, L and thus $L^\dagger$. Again, L' is readily determined after applying $\mathcal{A}$ to $\mathcal{I}_1$. The difficulty occurs in determining 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, position 1 and the reshaped access to array `A`, namely, `B` in position 2. Here

$$L = [1, 4] \quad \text{and} \quad L^{\dagger} = \begin{bmatrix} (\cdot)\%4 \\ (\cdot)/4 \end{bmatrix} \tag{52}$$

$$S = \begin{bmatrix} (\cdot)\%2 \\ (\cdot)/2 \end{bmatrix} \quad \text{and} \quad S^{\dagger} = [1, 2] \tag{53}$$

If we apply the permutation matrix

$$\mathcal{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{54}$$

then

$$L' = [1, 4] \tag{55}$$

As stated before, we have freedom with S'. If we choose $S' = S$, i.e., preserving the original shape of the array, we have the program in figure 1, position 6. As is immediately apparent, though correct, this access function to `B` will be extremely costly.

7.2. Reducing Access Overhead for Reshaped Arrays

In this subsection we examine two methods whereby loop transformations may be used to reduce any introduced overhead due to singular matrices. Just as in the case of linearised arrays, we can remove some of the overhead due to `mods` etc. by applying a loop transformation to remove some of those rank-increasing transformations which introduce `mods`. Let T be the loop transformation to remove these operators

$$T = \mathcal{U}_2^{-1} L^{\dagger} S^{\dagger} \mathcal{U}_2 \tag{56}$$

and hence

$$T^{-1} = \mathcal{U}_2^{-1} S L \mathcal{U}_2 \tag{57}$$

Applying such a transformation would have the following affect on the access $\mathcal{U}_2'$ from equation (51)

$$\mathcal{U}_2' T^{-1} = S' L' \mathcal{A} L^{\dagger} S^{\dagger} \mathcal{U}_2 T^{-1} = S' L' \mathcal{A} \mathcal{U}_2 \tag{58}$$

If this transformation is applied to the code in position 6 in figure 1, we produce the program shown in position 9, figure 2. Although this removes some of the expensive functions, there still remain `mods` due to S'. A possibly more straightforward approach would be to select the reshaping matrix S' to be $S' = L'^{\dagger}$. This has the

1. Given the original and reshaped array indices $\mathcal{I}_1$ and $\mathcal{I}_2$, 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. Let $T = \mathcal{A}L^\dagger$.

5. For each loop nest containing reference to the array, apply the loop transformation T, if it is legal to do so.

FIG. 6. Propagation Algorithm

effect that arrays are remapped into the same shape as the original array when applying a propagated data transformation $\mathcal{A}$. Thus equation (50) simplifies to:

$$\mathcal{I}_2' = S'L'\mathcal{I}_1 = S'L'\mathcal{A}\mathcal{I}_1 = \mathcal{A}L^\dagger S^\dagger \mathcal{I}_2 \tag{59}$$

If we applied this data transformation to the code in position 6 of figure 1 we arrive at the code in position 12, figure 2. Finally, a simple method to improve performance is to examine the array access matrix and then strip-mine those iterators which are arguments of `divs` or `mod`, and reorder the iterators to give the code in position 13, figure 2 with the stride-1 access shown in position 14, figure 2. In other words, apply a loop transformation $T = \mathcal{A}L^\dagger$ and update the loop bounds and array accesses accordingly.

8. EXPERIMENTS

In this section we develop a simple algorithm to eliminate the overhead associated with reshaped arrays and apply the technique to three SPEC benchmarks demonstrating the usefulness of our approach.

8.1. Algorithm

In this subsection, we develop an algorithm for propagating data transformations to reshaped arrays. Given the techniques developed in sections 5 and 6, it is relatively straightforward to incorporate then into a compiler phase order. The phase order of our compiler [5] is data alignment, data partitioning, synchronisation placement, loop partitioning, locality optimisation and code generation. Once the data alignment of each array is known, we can apply the algorithm in figure 6. Thus it is applied after data alignment has been chosen but before data partitioning. The determination of alignment is based on a heuristic described in [24] where conflicting requirements are weighted by the relative importance of the array accesses considered. Section 9 reviews alternative approaches to determining data alignment.

Consider the algorithm in figure 6. Once our compiler has determined the global data alignment, 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) before a loop transformation is constructed (step 4) to remove any remaining `mod`s etc (step 5). As step 5 may reorder the loop nest, its legality must be checked before application.

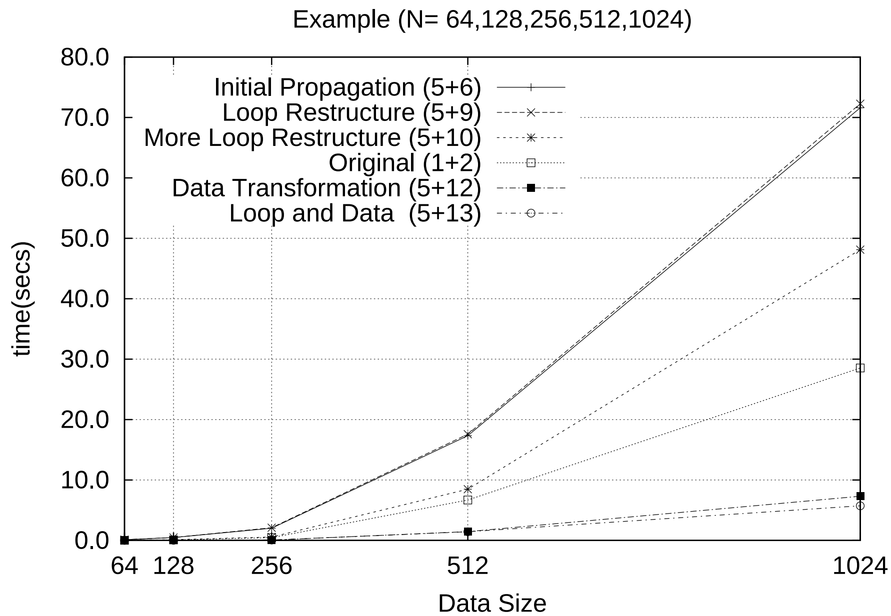


FIG. 7. Example Program: N = 64. . . 1024

This algorithm assumes that the layout of each actual array parameter passed as a formal parameter to a subroutine with a reshaped array is the same at each call site or that cloning has taken place.

8.2. Results

To show the use of the analysis developed in this section, we ran each of the program versions shown in figures 1 and 2 on the SGi Challenge for varying data sizes. Figure 7 shows their relative performance compared to the original program. Each line corresponds to different versions of the program (shown in figures 1 and 2) generated after applying various transformations. The numbers following the title of each line refer to the position number, in figures 1 and 2, containing the code evaluated. For instance in the case of initial propagation, the code executed is that of positions 5 and 6.

The basic data propagation scheme is more than twice as poor as the original. For this reason, it is not surprising that compiler writers disable data transformations in the presence of reshaped arrays. Although subsequent loop transformations do improve performance, they still do not 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 6 to three SPECfp92 benchmarks `vpetst, btrtst` and `chotst`. These program were selected as they each contain a subroutine call where array arguments are reshaped. In each case there is one or more calls to a copy routine where the actual array arguments are mapped to the linearised formal array argument of the copy subroutine. Each benchmark was parallelised by our compiler, MARS [5], and the new algorithm was applied by hand and compared to the existing technique of remapping array data

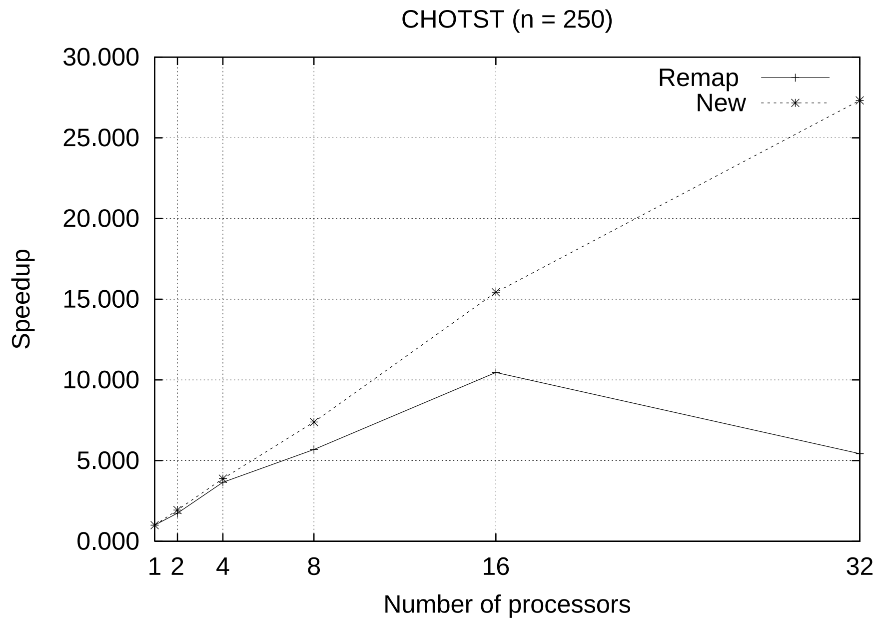


FIG. 8. CHOTST: N = 250

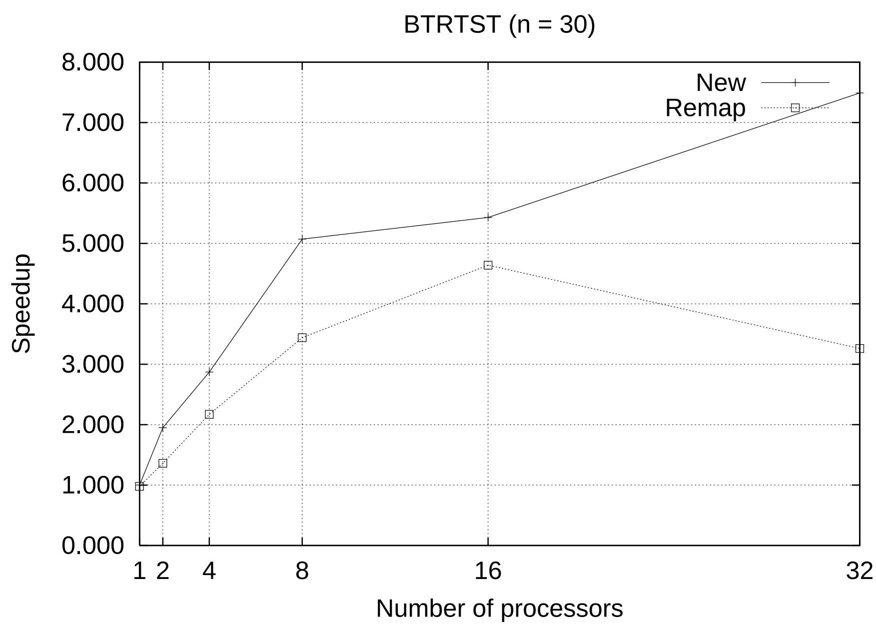


FIG. 9. BTRTST: N = 30

at run time. The two versions of each benchmark were executed on an SGi Origin 2000 with optimisation level -O3. The results are shown in figures 8, 9 and 10, where the x-axis corresponds to the number of processors P and the y-axis to the speedup achieved. MARS selected only simple permutations for data alignment

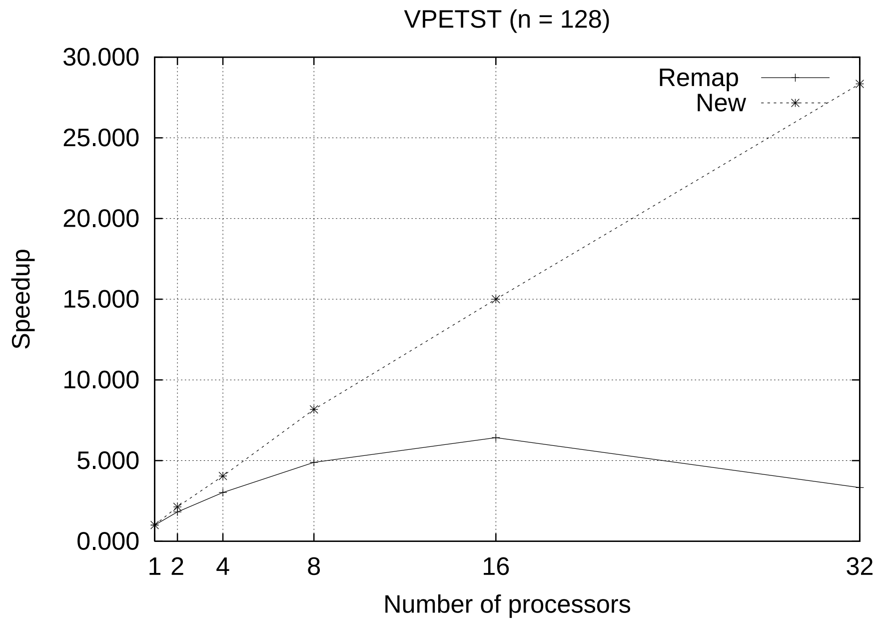


FIG. 10. VPETST: N = 128

and thus calculating the layout of the reshaped arrays was straightforward with no `mod` or `div` operators introduced.

For each benchmark, the relative performance of the code generated by the algorithm (labelled "New") increases with the number of processors when compared to the standard scheme (labelled "Remap"). This improvement varies from a factor of 3 in the case of `btrtst` to 10 in the case of `vpetst`. The variance is due to the amount of remapping relative to useful computation in each program and the data layout chosen by the compiler in order to exploit parallelism. For instance, in `chotst`, the amount of work per processor is $O(\frac{N^4}{P})$ while the amount of remapping is $O(N^3)$. As P increases, the relative overhead of remapping increases, eventually dominating when $P = 32$, causing a slow-down. In `btrtst`, the amount of work and remapping are of the same order of magnitude, but as the arrays are all partitioned on the last dimension, the linearised remapping within the subroutine does not incur any redistribution or non-local access. The only overhead is due to barrier synchronisations and thus there is only a small slow down in the case of $P = 32$. Finally, in the case of `vpetst`, once again the amount of work and remapping are of the same order of magnitude but, in this case, the arrays are partitioned along the first dimension. The linearised remapping within the subroutine effectively redistributes the array into an array partitioned along the last dimension incurring much non-local access. Hence the increased relative performance of our algorithm in this case and the slow down of the original code when $P = 32$. Therefore, we conclude, our algorithm improves programs in those cases where remapping of data is an important factor in execution time.

The values of N are fixed as part of the Spec benchmarks. If N were to increase, it would have no significant impact on the relative performance of our scheme with respect to the standard approach on `vpetst` and `btrtst` as the cost of computation and remapping is the same order of magnitude. However the absolute speedups obtained for both schemes would problem improve for `btrtst` as $N = 30$ is small and gives little scope for efficient parallelisation on a large number of processors. In the case of `chotst`, for a fixed processor number, the relative impact of remapping will decrease as N grows, but will still be a significant overhead.

9. RELATED WORK + DISCUSSION

There is a large body of work concerned with improving program performance using program transformations. In [25], unimodular loop transformations are used to improve parallelism and locality, which is extended to the non-singular case in [20]. These papers restrict themselves to loop transformations and do not consider cases where there are conflicting loop orders. In such instances, [21] proposes a simple heuristic, limited to just loop permutations, which is extended to the unimodular case in [3]. Although useful, these approaches are limited in that they do not consider array layout transformations.

Other researchers have considered data transformations, primarily with respect to data alignment and partitioning. In [2, 9, 14, 19] approaches based on graph theory, integer programming and linear algebra are explored. Most of this work, however, considers alignment to be part of a mapping process rather than a program transformation and thus, parallels with loop transformation are absent. In [22], alignment as a program level transformation is first presented while [18] uses a similar representation for uni-processor spatial locality. In [6], data linearisation transformations are considered as a means to change array layout, while data permutation and strip-mining transformations are considered in [1]. In [23], we developed a new framework describing non-singular data transformations equivalent in standing to loop transformations and described how they may be used to improve program performance. Again, these approaches are limited as they restrict themselves to just data transformations.

There has been recent work considering the combination of loop and data transformations in improving program performance. Using a hyperplane formulation of data transformations and non-singular loop transformations, an algorithm, which considers a restricted set of loop and data transformations, is proposed as a means of improving locality [6]. In [11, 12], a similar formulation is used, but considers a wider class of transformations. These approaches, however, have an asymmetric treatment of loop and data transformations. Furthermore, transformations such as iteration space [13] and data space tiling have not been integrated into the above work. In [1], data tiling is used to improve spatial locality, but the representation used does not allow easy integration with other data space transformations or general loop transformations. In this paper, we overcome this problem by developing an extended transformation framework based on rank modifying transformations.

In [7], 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 [7].

This paper describes a technique to allow propagation of data transformations in the presence of reshaping. As reshaping primarily occurs across procedure boundaries, a full implementation would require integration within an inter-procedural analysis phase. In particular the technique is calling context sensitive, i.e. the transformations propagated to the subroutine depend on where it is called from, thus there is a possibility of conflict when there are two or more caller-sites. In such circumstances, the compiler may decide not to propagate one or more of the data transformations or may decide to clone the called subroutine.

Although this paper has focused on the use of rank-modifying transformations as a means to allow data transformation propagation, they do have wider application. They can model the effect of linearisation and strip-mining on loops and arrays and when combined with interchange can represent loop tiling or block-cyclic data partitioning. Future work will investigate the applicability of rank-modifying transformations in context other than array reshaping.

10. CONCLUSION

In this paper, we have presented a new algebraic framework that allows the integration of loop and data transformations. This enables existing transformations to be described in a unifying manner and provides also the basis for new program optimisations. In particular, we develop 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.

There are, however, many other uses of such a framework in auto-parallelisation and locality optimisation. The central problem of loop or data only approach to parallelisation is that of balancing conflicting requirements throughout the program. More specifically, if one part of a program, be it a loop or an array access, requires a particular transformation but another part requires a completely distinct transformation, how do we determine transformations that trade-off such requirements to give a globally acceptable result? Future work will investigate how conflicting requirements on a loop transformation may be resolved by using a data transformation and vice-versa. Future work will also further investigate the properties of the transformation representation used in this paper and optimisation algorithms.

REFERENCES

1. J.M. Anderson, S.P. Amarasinghe and M.S. Lam, Data and computation transformations for multiprocessors, *in* "Proc. 1995 Principles and Practice of Parallel Programming," ACM Press, pp. 166-178, 1995.

2. D. Bau, I. Kodukula, V. Kotlyar, K. Pingali and P. Stodghill, Solving alignment using elementary linear algebra, *in* "Proc Seventh Annual workshop on Languages and Compilers for Parallel Computing", LNCS 892, Ithaca, New York, pp. 46-60, 1994.

3. A.J.C. Bik and P.M.W. Knijnenburg, Reshaping access patterns for improving data locality, *in* "Proc. Sixth Workshop on Compilers for Parallel Computers," pp. 299-310, Aachen, 1996.

4. U. Banerjee, "Loop Transformations for Restructuring Compilers," Kluwer Academic Publishers, Norwell, 1993.

5. F. Bodin and M.F.P. O'Boyle, A compiler strategy for SVM, *in* "Proc. Third Workshop on Languages, Compilers and Runtime Systems for Scalable Computers" (B.K. Szymanski and B. Sinharoy, Eds.), Kluwer Academic Publishers, Norwell, MA, pp. 57-69, 1995.
6. M. Cierniak and W. Li, Unifying data and control transformations for distributed shared-memory machines, *in* "Proc. 1995 Programming Language Design and Implementation," ACM Press, pp. 205 - 217, 1995.
7. M. Cierniak and W. Li, "Validity of Interprocedural Data Remapping," Tech. Rep. 642, University of Rochester, Rochester, 1996.
8. P. Feautrier, Dataflow Analysis of scalar and array references in "International Journal of Parallel Programming" 20(1) pp. 23-53 , February 1991.
9. M. Gupta, "Automatic Data Partitioning on Distributed-Memory Multicomputers," Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.
10. T.E.Jeremiassen and S.J. Eggers, Reducing false sharing on shared memory multiprocessors through compile-time data transformations, *in* "Proc. Principles and Practice of Parallel Programming," ACM Press, pp. 179 - 188, 1995.
11. M. Kandemir, J. Ramanujam and A. Choudhary, A compiler algorithm for optimizing locality in loop nests, *in* "Proc. 11th ACM International Conference on Supercomputing," pp. 269-276, ACM Press, 1997.
12. M. Kandemir, A. Choudhary, J. Ramanujam and P.Banerjee, A matrix approach to the global locality optimization problem, *in* "Proc. International Conference on Parallel Architectures and Compilation Techniques," pp. 306-313, IEEE Computer Society, Paris, 1998.
13. I. Kodukula, N. Ahmed and K. Pingali, Data-centric multi-level blocking, *in* "Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation", ACM Press, pp. 346 - 357, 1997.
14. K. Kennedy and U. Kremer, Automatic data layout for High Performance Fortran, *in* "ACM Transactions on Programming Languages and Systems", 20(4), pp. 869-916, 1998
15. K. D. Cooper, M. W. Hall and K. Kennedy, "A Methodology for Procedure Cloning" Computer Languages, 19(2), pp. 105- 117, April 1993
16. D. Kulkarni and M. Stumm, "Loop and Data Transformations: A Tutorial," Tech. Rep. CSRI-337, University of Toronto, Toronto, 1993.
17. P.M.W. Knijnenburg, E. Ayguadé and J. Torres, "Multi-Transformations of Nested Loops for Parallelizing Compilers," Tech. Rep. 96–14, Leiden University, Leiden, 1996.
18. S.-T. Leung and J. Zahorjan, "Optimizing Data Locality by Array Restructuring," Tech. Rep. 95-09-01, Department of Computer Science and Engineering, University of Washington, 1995.
19. J. Li and M. Chen, Index domain alignment: Minimising cost of cross-referencing between distributed arrays, *in* "Proc. of the Third Symposium on the Frontiers of Massively Parallel Computation," IEEE Computer Society, 1990.
20. W. Li and K. Pingali, A singular loop transformation framework based on non-singular matrices, *in* "International Journal of Parallel Programming" 22(2), pp. 183-205, 1994.
21. K. McKinley, S. Carr and C.-W. Tseng, Improving data locality with loop transformations, *in* "ACM Transactions on Programming Languages and Systems", 18(4), pp. 424-453, 1996.
22. M.F.P. O'Boyle and G.A. Hedayat, Data alignment: Transformation to reduce communication on distributed memory architectures, *in* "Proc. Scalable High Performance Computing Conference," IEEE Computer Society, Virginia, pp 366 - 371, 1992.
23. M.F.P. O'Boyle and P.M.W. Knijnenburg, Non-Singular Data Transformations: Definition, Validity and Applications, *in* "International Journal of Parallel Programming", 17(3), pp. 131-159, 1999.
24. M.F.P. O'Boyle and P.M.W. Knijnenburg Efficient Parallelisation using Combined Loop and Data Transformations, *in* "Parallel Architectures and Compiler Technology", IEEE Press, Newport Beach, pp. 283 -293, October 1999.
25. M.E. Wolf and M. Lam, A Data Locality Optimizing Algorithm *in* "Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation", pp. 30 -44, June, 1991

APPENDIX

Proof 2

1.To show that $\mathbf{L}^\dagger$ is a proper inverse of $\mathbf{L}$, we need to show that $\mathbf{L}^\dagger(\mathbf{L}(a)) = a$ where a is a 2 dimensional vector defined over a sub-space D^2 of $\mathbf{Z}^2$. a corresponds to a 2 dimensional index vector or iterator and $D^2 = (0 : n_1 - 1, 0 : n_2 - 1)$ corresponds to the index/iteration space over which a traverses.

$\mathbf{L}^\dagger(\mathbf{L}(a)) = \begin{bmatrix} (\cdot)\%n_1 \\ (\cdot)/n_1 \end{bmatrix} \begin{bmatrix} 1 & n_1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$

which reduces to

$\begin{bmatrix} (\cdot)\%n_1 \\ (\cdot)/n_1 \end{bmatrix} \begin{bmatrix} a_1 + n_1 a_2 \end{bmatrix} = \begin{bmatrix} (a_1 + n_1 a_2)\%n_1 \\ (a_1 + n_1 a_2)/n_1 \end{bmatrix}$

As $a_1 < n_0$ then $a_1\%n_0 = a_1$ and $(a_1 + n_1 a_2)\%n_1 = a_1\%n_1 + (n_1 a_2)\%n_1 = a_1 + 0$

Expanding, $(a_1 + n_1 a_2)/n_1 = a_1/n_1 + (n_1 a_2)/n_1 + (a_1\%n_1 + n_1 a_2\%n_1)/n_1$

This reduces to $0 + a_2 + (a_1 + 0)/n_1 = 0 + a_2 + 0 = a_2$

Thus $\mathbf{L}^\dagger(\mathbf{L}(a)) = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = a$

2. Here we must show that $\mathbf{S}^\dagger(\mathbf{S}(a)) = a$ where a is a 1-dimension vector corresponding to an index or iteration vector.

$\mathbf{S}^\dagger(\mathbf{S}(a)) = \begin{bmatrix} 1 & n_1 \end{bmatrix} \begin{bmatrix} (\cdot)\%n_1 \\ (\cdot)/n_1 \end{bmatrix} [a] = [(\cdot)\%n_1 + n_1 * ((\cdot)/n_1)][a]$

This transformation maps an integer a to $a\%n_1 + n_1 * (a/n_1)$. Now for any $a \in \mathbf{Z}$, $a\%n_1 + n_1 * (a/n_1) = a$ and $\mathbf{S}^\dagger(\mathbf{S}(a)) = a$

Proof 3

We have $L = \begin{bmatrix} I_{k-1} & O & O \\ O & \mathbf{L} & O \\ O & O & I_{N-k} \end{bmatrix}$ and $L^\dagger = \begin{bmatrix} I_{k-1} & O & O \\ O & \mathbf{L}^\dagger & O \\ O & O & I_{N-k} \end{bmatrix}$

Using theorem 2 we have $L^\dagger \times L = \begin{bmatrix} I_{k-1} & O & O \\ O & \mathbf{L}^\dagger\mathbf{L} & O \\ O & O & I_{N-k} \end{bmatrix} = I$

Proof 4

Consider the loop linearisation transformation $L = \begin{bmatrix} I_{k-1} & O & O \\ O & \mathbf{L} & O \\ O & O & I_{N-k} \end{bmatrix}$

All entries apart from the $\mathbf{L}$ term are identity mappings from the original domain to the new one. Thus without loss of generality we examine the action of the $\mathbf{L}$ matrix on a two dimension iteration vector.

Let the original iteration vector $J = \begin{bmatrix} i & j \end{bmatrix}^T$ range over the domain $(0 : n_1 - 1, 0 : n_2 - 1)$ and the new vector be denoted by $J' = \begin{bmatrix} i' \end{bmatrix}$. To demonstrate that the new loop after applying $\mathbf{L}$ is legal we need to show firstly that the order of the iterations is preserved in the new domain and secondly that the for any iteration point in the original domain the array element accessed is the same for the corresponding iteration point in the new domain.

1. We need to show that lexicographic order is preserved wrt. to the linearisation transformation. Given two points in the original domain $(i_1, j_1), (i_2, j_2)$ and two in the new domain (i'_1, i'_2) we need to show that $i_1, j_1 \prec i_2, j_2$ implies that $i'_1 \prec i'_2$ [8].

If $i_1, j_1 \prec i_2, j_2$ then either (a) $i_1 < i_2$ or (b) $i_1 = i_2 \wedge j_1 < j_2$ must be true [8]. We show that lexicographic ordering is preserved under both circumstances.

(i) $n_1 i + j = i'$ from the definition of **L**, thus we must show that $n_1 i_1 + j_1 < n_1 i_2 + j_2$. We show that the maximum value of the LHS is always smaller than the minimum value of the RHS and thus the inequality always holds.

RHS: Now as $i_1 < i_2$ then the smallest value of i_2 for which this is true is $i_1 + 1 = i_2$. Similarly, the smallest value of j_2 is 0.

LHS :The largest value of j_1 is $n_1 - 1$. and thus the inequality can be rewritten $n_1 i_1 + n_1 - 1 < n_1 i_1 + n_1 + 0$ which is always true.

(ii) Here we must show that $n_1 i_1 + j_1 < n_1 i_1 + j_2$. Now as $j_1 < j_2$, the result immediately follows.

2. We need to show that an array access in the old domain a(x_1, y_1) and an access in the new domain a(x_2, y_2) refer to the same points when $n_1 i + j = i'$. Without loss of generality, let us assume that the original array access was of the form `a(i,j)` and thus the new access is of the form `a(i'/ n, mod (i',n))`. We need to show that $i = i'/n_1$ and $j = i'\%n_1$ when $n_1 i + j = i'$.

Now $i'/n_1 = (n_1 i + j)/n_1 = n_1 i/n + j\%n_1 + (n_1 i\%n_1 + j\%n)/n_1$. This reduces to $i + 0 + (0 + j\%n_1)/n_1$ as $j < n_1$ and thus $j\%n_1 = 0$.

By definition $j\%n_1 < n_1$ and thus $j\%n_1/n_1 = 0$. Therefore $i = i'/n_1$.

Similarly, $i'\%n_1 = (n_1 i + j)\%n_1 = n_1 i\%n_1 + j\%n_1 = 0 + j\%n_1$. As $j < n_1$ we have that $j = i'\%n$.

Therefore loop order is preserved and corresponding iteration points access the same array elements.

A similar argument holds for **S**

Proof 5

Data transformations are simply a generalised method for renaming array elements and are always legal in programs without subroutines if there is a one to one mapping of elements to the new domain [6, 23, 11] In the case of 2 or more calls to a subroutine where one of the callee sites has a transformed array as one of its actual arguments then there are difficulties in applying general non-singular data transformations . However, in the case of linearisation or strip-mining the underlying layout in machine linear memory is unaffected, merely the index domain defined in the array declaration. As the underlying machine memory layout is unaffected, then it has no impact on the callee subroutine. Hence rank-modifying data transformations are always legal.

MICHAEL O'BOYLE received a PhD. in Computer Science from the University of Manchester in July 1992. He was formerly a SERC Postdoctoral Research Fellow, a Visiting Research Scientist at IRISA, Rennes and a Visiting Research Fellow at the Institute of Software Development and Parallel Systems, University of Vienna. He is currently an EPSRC Advanced Research Fellow, a lecturer in the Institute for Computing Systems Architecture (ICSA) at the University of Edinburgh where he is a member of the Compiler group and an honorary lecturer at the University of Manchester. His main research interests are in automatic compilation for distributed shared memory architectures, adaptive and iterative compilation for embedded systems and compiler directed memory coherence.

PETER M.W. KNIJNENBURG received an M.Sc. in physics and an M.Sc. in computer science from Utrecht University, the Netherlands, in 1988. Subsequently, he received a Ph.D. in computer science from Utrecht University in September 1993. He is currently an assistant professor in computer science at Leiden University in the Netherlands. His main interests are in compilation for Instruction Level Parallel processors, computer architecture, adaptive compilation for embedded systems and the formal representation of code and data transformations.