Computational Alignment: A New, Unified Program Transformation for Local and Global Optimization

Dattatraya Kulkarni and Michael Stumm

Technical Report CSRI-292
January, 1994

Computer Systems Research Institute,
Department of Computer Science, and
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 1A4

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Computational Alignment:
A New, Unified Program Transformation for Local and Global Optimization

Dattatraya Kulkarni and Michael Stumm

Department of Computer Science, and
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada, M5S 1A4
Email: kulki@csri.toronto.edu

January, 1994

Abstract

Computational Alignment is a new class of program transformations suitable for both local and global optimization. Computational Alignment transforms all of the computations of a *portion* of the loop body in order to align them to other computations either in the same loop or in another loop. It extends along a new dimension and is significantly more powerful than linear transformations because *i*) it can transform subsets of dependences and references; *ii*) it is sensitive to the location of data in that it can move the computation relative to data; *iii*) it applies to imperfect loop nests; and *iv*) it is the first loop transformation that can change *access vectors*. Linear transformations are just a special case of Computational Alignment. Computational Alignment is highly suitable for global optimization because it can transform given loops to access data in similar ways. Two important subclasses of Computational Alignment are presented as well, namely, *Freeing* and *Isomerizing* Computational Alignment.

1 Introduction

Optimizing compilers for parallel machines transform source programs in order to improve performance on target parallel hardware. They map computations and data onto processors to extract parallelism and minimize data movement. From early on, nested loops have been recognized as a major source of parallelism. Consequently, there is a large body of work on transforming nested loops [1, 2, 3, 4, 5, 6, 7, 8] and the arrays they access [9, 10, 11] for the purpose of improving locality, parallelism and load balance. Some of the basic loop transformations are interchange, skew, reversal, wavefront, permutation, and tiling [1, 2, 3, 4, 6, 12, 13, 14, 15, 16]. A data transformation applied often is data alignment, which maps one array onto another.

In practice, when optimizing a loop, it is generally necessary to apply a sequence of several transformations. Deriving the appropriate sequence of transformations is non-trivial and early work in the area did not address this issue. *Linear loop transformations* [17, 18, 19, 20] provide a unifying framework in which a sequence of transformations is represented by a non-singular matrix. The algebraic framework allows us to derive the transformed loop structure in a systematic way in one step from the given transformation matrix and the original loop structure. In our earlier work, we posed the problem of finding the best linear transformation of nested loops as an optimization problem, and presented techniques to obtain near optimal solutions [19, 21, 22]. The framework of linear transformations is very general; even data alignment can be thought of as a linear transformation.

In this paper, we present a new class of linear transformations that extends traditional linear loop transformations along a new dimension. This new class of transformations is called *Computational Alignment* and is significantly more powerful than the existing linear loop transformations. In fact, traditional linear loop transformations are just a special case of Computational Alignment in our framework. Computational Alignment enables us to perform *local optimizations* that were not possible with earlier techniques. In addition, it is highly suitable for *global optimizations* [23, 11, 24, 25]. In contrast, existing loop and data transformations only consider constraints relative to a single loop, and can therefore only be applied as local optimizations.

Computational Alignment is a transformation of all the computations associated with a *portion* of the loop body in order to align them to other computations either in the same loop or in another loop. This provides us with an additional degree of freedom because linear loop transformations always transform the loop body as a whole. Because of this additional degree of freedom, Computational Alignment *i*) can transform subsets of dependences and references; *ii*) is sensitive to the location of data in that it can move the computation relative to data; *iii*) applies to imperfect loop nests; and *iv*) is the first loop transformation that can change *access vectors*. Because of these capabilities, Computational Alignment bridges local optimization techniques with global optimization techniques.

Section 2 provides an overview of existing techniques in linear loop transformations and data alignments. The section is tutorial in nature and a reader familiar with the material is encouraged to skip it. We provide an overview of Computational Alignment and summarize its features in Section 3. Computational Alignment is formally defined in Section 4. This section also describes how to derive the transformed structure of computationally aligned loops. The basic algorithms presented have the disadvantage that they require the introduction of *guards* to control the set of computations being executed. Since this adds extra run-time overhead to the computation, we present two algorithms in Section 7: one reduces the number of guards required; the other eliminates the need for guards entirely, but at the expense of potentially transforming a perfectly nested loop into an imperfectly nested one. We close the paper with a discussion of the related and future work.

2 Loop and Data Transformations

In this section we summarize the existing techniques for linear loop transformations and data alignments. Before doing so, we briefly describe the underlying algebraic framework. The program model is that of an affine nested loop:

$$
\begin{array}{l}
\texttt{for}\ \ \mathtt{I_1 = l_1, u_1} \\
\quad \texttt{for}\ \ \mathtt{I_2 = l_2(I_1), u_2(I_1)} \\
\qquad \ldots \\
\qquad\quad \texttt{for}\ \ \mathtt{I_n = l_n(I_1,..,I_{n-1}), u_n(I_1,..,I_{n-1})} \\
\qquad\qquad \mathtt{H(I_1,...,I_n)} \\
\qquad\quad \texttt{end for} \\
\qquad \ldots \\
\quad \texttt{end for} \\
\texttt{end for}
\end{array}
$$

$I_1, ..., I_n$ are the iteration indices; l_i and u_i are the lower and upper loop limits, which are linear functions of iteration indices $I_1, .., I_{i-1}$; and implicitly a stride of one is assumed. $\mathtt{I}= (I_1, ..., I_n)^T$ is called the iteration vector and denotes an iteration of the loop. The same iteration vector is also denoted by $\mathtt{I}= (I_1, ..., I_n, 1)^T$ when we use a homogeneous co-ordinate system. H is the body of the nested loop. Typically, an access to an m-dimensional array `A` in the body of the loop has the form $\mathtt{A}(f_1(I_1, ..., I_n), ..., f_m(I_1, ..., I_n))$, where the f_i 's are affine functions of the iteration indices, which means that an array reference can be represented by a $m \times (n + 1)$ matrix called the *reference matrix* in homogeneous co-ordinates.

The iteration space corresponds to the integer space described by the loop bounds, and is defined as [2]:

Definition 1 (Iteration space) *An iteration space is a set $\mathcal{I} \subseteq \mathbf{Z}^n$ such that*

$$\mathcal{I} = \{(i_1, ..., i_n) \mid l_1 \leq i_1 \leq u_1, ..., l_n(i_1, ..., i_{n-1}) \leq i_n \leq u_n(i_1, ..., i_{n-1})\},$$

where i_1, ..., i_n are the iteration indices, and $(l_1, u_1), ..., (l_n, u_n)$ are the respective loop limits.

The iteration space is a convex polyhedron and the *bounds matrix* is a succinct way of specifying the bounds of this space. Since each of the lower and upper bounds are affine functions of the iterators, the set of bounds can be written in a matrix vector notation. The set of lower bounds can be represented by

$$\beta_L \cdot \mathtt{I} \geq \mathtt{l}$$

where β_L is an $n \times n$ integer lower triangular matrix, $\mathtt{I}$ is the $n \times 1$ iteration vector, and $\mathtt{l}$ is a $n \times 1$ integer vector. Similarly, the upper bounds can be represented by

$$\beta_U \cdot \mathtt{I} \leq \mathtt{u} \quad \text{or} \quad -\beta_U \cdot \mathtt{I} \geq -\mathtt{u}$$

where β_U is an integer lower triangular matrix and $\mathtt{u}$ is a $n \times 1$ integer matrix. Both the lower and upper bounds can be represented by a single matrix β.

$$\beta \cdot \mathtt{I} \geq \mathtt{c}$$

where

$$\beta = \begin{bmatrix} \beta_{\mathtt{L}} \\ -\beta_{\mathtt{U}} \end{bmatrix}, \quad \mathtt{c} = \begin{bmatrix} \mathtt{l} \\ -\mathtt{u} \end{bmatrix}$$

For convenience, we represent this inequality in the homogeneous co-ordinate system as

$$\begin{bmatrix} \beta & -\mathtt{c} \end{bmatrix} \cdot \begin{bmatrix} \mathtt{I} \\ 1 \end{bmatrix} \geq \begin{bmatrix} \mathtt{0} \end{bmatrix}$$

The matrix $\begin{bmatrix} \beta & -\mathtt{c} \end{bmatrix}$ is called the bounds matrix and since the context of homogeneous co-ordinates is clear, we denote it by β.

Data dependences in the loop nest are precedence constraints on iterations. When the dependences are uniform across the iteration space, the dependence relation can be reprenseted by a vector of integers. For a pair of iterations $\mathtt{i} = (i_1, \ldots, i_n)$ and $\mathtt{j} = (j_1, \ldots, j_n)$ such that $\mathtt{j}$ depends on $\mathtt{i}$, the vector $\mathtt{j}$ -$\mathtt{i}$ $= (j_1 - i_1, ..., j_n - i_n)$ is called the *dependence distance vector*. A dependence $(d_1, .., d_n)$ is said to be lexicographically positive or simply positive, if the first nonzero element d_k is positive, and we then say that the dependence is carried by the k^{th} loop.

The dual of the dependence distance vector is the *access vector* in the data space of the array. If $\mathtt{d}$ is the dependence between two references A_1 and A_2 to an array $\mathtt{A}$:

$$A_1 = \left[\begin{array}{c|c} V\ n \times n & \mathtt{d_1}\ {}_{n \times 1} \\ \hline 0\ {}_{1 \times n} & 1 \end{array}\right] \quad A_2 = \left[\begin{array}{c|c} V\ n \times n & \mathtt{d_2}\ {}_{n \times 1} \\ \hline 0\ {}_{1 \times n} & 1 \end{array}\right]$$

then the access vector $\mathtt{a}$ is:

$$\mathtt{a} = V \cdot \mathtt{d}$$

The access vector identifies the direction in the array along which a set of dependent iterations access the data.

Example 1 *:* **Model of Loop nest**
Consider the following double loop.

```
for  I1 = 0, 10
  for  I2 = 0, 10
    S1 :  A(I1, I2) = A(I1 + I2 + 2, I2 - 1) + C
    S2 :  B(I1, I2) = A(I1 - 1, I2 - 2) + D
  end for
end for
```

The reference $\mathtt{A}(\mathtt{I}_1 + \mathtt{I}_2 + 2, \mathtt{I}_2 - 1)$, can be represented by the reference matrix and the iteration vector:

$$\begin{bmatrix} 1 & 1 & 2 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ 1 \end{bmatrix}$$

The bounds of the loop are represented by the bounds matrix and iteration vector:

$$\left[\begin{array}{rrr} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \hline -1 & 0 & 10 \\ 0 & -1 & 10 \end{array}\right] \begin{bmatrix} I_1 \\ I_2 \\ 1 \end{bmatrix} \geq \left[\begin{array}{c} 0 \\ 0 \\ \hline 0 \\ 0 \end{array}\right]$$

The only uniform dependence is a flow of data from S1 to S2 and is denoted by $(1, 2)$. The corresponding access vector is $(1, 2)$ as well. $\square$

2.1 Linear loop transformations

Most of the loop transformations proposed to date can be formalized as linear transformations of the iteration space, which reorganize the iterations in the loop nest to produce the desired loop and dependence structures. The introduction of linear loop transformations [17, 18, 19, 20] was a major step in unifying most existing loop transformations. In this framework, a non-singular integer matrix completely characterizes the transformation. That is, the loop bounds, dependences, and references of the transformed loop can be computed directly from the transformation matrix, and the original bounds matrix, dependences, and reference matrices. The framework is unifying in that, any sequence of linear transformations such as reversal, interchange, and skew [1] can be combined and represented by a single linear transformation.

Suppose $\mathtt{I}$ is the iteration vector, β the bounds matrix, D the *dependence matrix* where each column is a dependence distance vector, and R an array reference in an n-dimensional nested loop. Suppose U is a $n \times n$ non-singular integer matrix representing a linear transformation, then the corresponding iteration vector $\mathtt{I}'$, dependence matrix D', and array reference R' of the transformed loop become:

$$\mathtt{I}' = U \cdot \mathtt{I}$$

$$D' = U \cdot D$$

$$R' = R \cdot U^{-1}$$

An access vector $\mathtt{a} = V \cdot \mathtt{d}$ however remains unchanged, because in the transformed loop :

$$\mathtt{a}' = V' \cdot \mathtt{d}' = V \cdot U^{-1} \cdot U \cdot \mathtt{d} = \mathtt{a}$$

Suppose we pad U with zeros in the $(n+1)^{th}$ column and row except for $U_{(n+1)(n+1)}$ which is set to 1. The original bounds matrix expressed a polyhedron:

$$\beta \cdot \mathtt{I} \geq \mathtt{0}$$

Since

$$\beta \cdot U^{-1} \cdot U \cdot \mathtt{I} \geq 0$$

we have

$$\beta \cdot U^{-1} \cdot \mathtt{I}' \geq 0$$

Thus, the new bounds matrix is $\beta \cdot U^{-1}$. Because the limits of a loop have to be expressed as functions of only the enclosing loop indices, if any at all, and $\beta \cdot U^{-1}$ may not conform to this requirement, we can apply the Fourier-Motzkin variable elimination technique [26] to $\beta \cdot U^{-1}$ to obtain the bounds matrix β' for the transformed loop. When U is not unimodular the transformed loop will have non-unit strides in some dimensions and extended Fourier-Motzkin variable elimination [27, 28] may have to be applied to get the exact transformed bounds.

Example 2 : Linear loop transformation
Consider the following double loop:

```
for  I1 = 0, 10
  for  I2 = 0, 10
    A(I1, I2) = A(I1 - 1, I2) + A(I1, I2 - 1)
  end for
end for
```

The dependence matrix for this nested loop is $D = \{(1,0),(0,1)\}$ Let the transformation (in homogeneous co-ordinates) be

$$U = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The original bounds matrix is

$$\beta = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 10 \\ 0 & -1 & 10 \end{bmatrix}$$

The new bounds matrix is then $\beta' = \beta \cdot U^{-1}$, i.e.

$$\beta' = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ -1 & 1 & 10 \\ 0 & -1 & 10 \end{bmatrix}$$

Since the new bounds matrix is not in the desired form, we apply the Fourier-Motzkin variable elimination. From β', it is clear that:

$$0 \leq I_2' \leq 10 \text{ and } I_1' - 10 \leq I_2' \leq I_1'$$

Thus the bounds for I_2' are:

$$max(I_1' - 10, 0) \leq I_2' \leq min(I_1', 10)$$

Once I_2' is eliminated, the above inequalities provide the following projections on I_1'.

$$I_1' - 10 \leq 10,\ 0 \leq I_1',\ 0 \leq 10,\ \text{and} I_1' \leq I_1' + 10$$

Ignoring the redundant constraints, we have constant bounds for I_1'.

$$0 \leq I_1' \leq 20$$

Thus the transformed program becomes:

```
for  I'1 = 0,20
  for  I'2 = max(0,I'1 − 10),min(10,I'1)
    A(I'1 − I'2,I'2) = A(I'1 − I'2 − 1,I'2) + A(I'1 − I'2,I'2 − 1)
  end for
end for
```

Figure 1 depicts the transformed iteration space. Dependences in the transformed loop are $(1, 0)$ and $(1, 1)$. Note that the iterations along the I_2' axis are all independent and can be executed in parallel. However, although we have transformed the loop to expose the parallelism, the amount of computation in each outer iteration varies greatly (from 1 to n inner iterations), so the load balance was made worse in the process. Hence, it is evident that the choice of transformation is a delicate trade-off among numerous parameters. □

The key contribution of linear loop transformations is that a linear transformation can represent a compound transformation of many existing transformations, and that the techniques to transform a loop are generic in that they are independent of the choice of a particular transformation.

2.2 Data alignment

The performance of a parallel program is dependent as much on the placement of data as it is on the structure of loops. Ideally, data should be local to the processor that will access it in order to minimize latency. The objective in aligning two arrays is to reorganize the data in one array with respect to that in the other in order to improve locality, assuming the arrays will be distributed the same way. O'Boyle and Hedayat [10] formalize alignment of a pair of arrays and provide a heuristic algorithm to do so. Li and Chen [11] provide a heuristic algorithm for the much more general task of aligning a group of arrays in a complete program.

An alignment of an array `B` to array `A` maps the elements of `B` to the elements of `A`. An identical distribution of the aligned arrays can result in reduced communications. If we

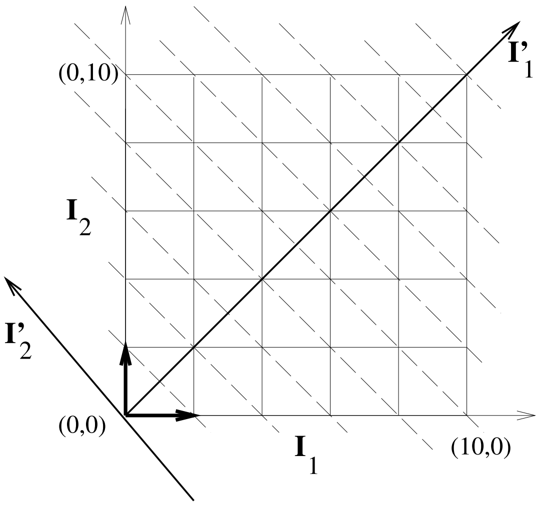
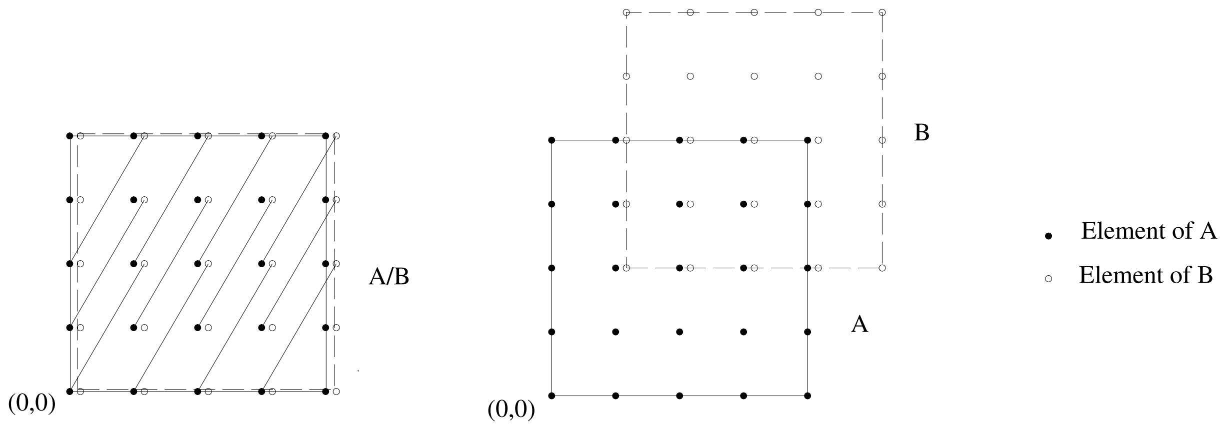


Figure 1: Example Loop Transformation



Before aligning B to A

After aligning B to A by (1,2) shift

Figure 2: Example Data Transformation

consider each element of array A to be on a different processor, then the alignment determines the elements of A and B that are assigned to the same processor. If we assume the familiar *ownership-rule* [29, 30] to determine which processor execute which computations, then an alignment will implicitly determine the placement of the computations in an iteration onto the processors.

Given two array references in a loop, the basic idea in finding an alignment transformation is to find the set of iterations in which elements of the arrays are indexed similarly. We then construct a transformation on one of the arrays such that the size of this set is maximum. This transformation can be represented by an integer matrix.

Example 3 *:* Data alignment
Consider, the following loop.

```
for  i = 1,n
  for  j = 1,n
    A(i,j) = B(i - 1,j - 2)
  end for
end for
```

Assume that arrays A and B are of the same size and B(i,j) is aligned to A(i,j), and each pair of elements (i,j) belongs to a processor. Then the resulting communication patterns are depicted in Figure 2, left. There are $O(n^2)$ communications. If, on the other hand, array B is remapped such that element B(i-1,j-2) is aligned to A(i,j), then the computation becomes communication free as long as A and B are distributed the same way (Figure 2, right). The transformed loop corresponds to:

```
for  i = 1,n
  for  j = 1,n
    A(i,j) = B(i,j)
  end for
end for
```

□

3 Computational Alignment: Overview

Computational Alignment is a new class of transformations that is similar to linear loop transformation in that it retains the similar algebraic framework. The key difference, however, is that Computational Alignment can transform a *portion* of a loop body independently instead of having always to transform the entire loop body, therefore be used to align computations corresponding to the portion of the loop to other computations in either the same loop or another loop. Because Computational Alignment can independently reorganize computations of any granularity (from individual assignment statements to entire loop nests) it effectively extends the linear loop transformations along a new dimension. In fact, Computational Alignment is unifying in the sense that any traditional loop transformation is just a special case of Computational Alignment where the computations to be reorganized are entire iterations.

In this section, we present two simple examples that illustrate the basic idea behind Computational Alignment and demonstrate its effectiveness in the context of both local and global optimizations. We follow the examples, by a summary of the key capabilities of the new transformation.

3.1 Simple Examples

As an example of what Computational Alignment can achieve locally, consider the following loop:

```
L :  for  i = 0,n
       for  j = 0,n
         S1 :  A(i,j) = A(i - 1,j - 1) + B(i,j - 1)
         S2 :  B(i,j) = A(i - 1,j) + C(i,j + i)
       end for
     end for
```

The loop has two dependences with respect to array A, namely (1,1) and (1,0), and a dependence with respect to array B, namely (0,1). Computational Alignment can shift all instances of statement S_2 to the left (i.e., in the $-i$ direction) by one in order to align the computation of $B(i+1,j)$ to the computation of $A(i,j)$, thus eliminating the (1,0) dependence. This results in the following loop (ignoring changes to the loop bounds for the moment):

```
L' :  for  i = 0,n
        for  j = 0,n
          S1 :  A(i,j) = A(i - 1,j - 1) + B(i,j - 1)
          S2 :  B(i + 1,j) = A(i,j) + C(i + 1,j + i)
        end for
      end for
```

Figure 3 illustrates the transformation. The top half of the figure shows the original loop structure (L) and the transformed loop structure (L$'$). The bottom half of the figure illustrates the same structures with the iteration spaces for S_1 and S_2 shown explicitly. A hollow square represents an S_1 computation and a filled square represents an S_2 computation.

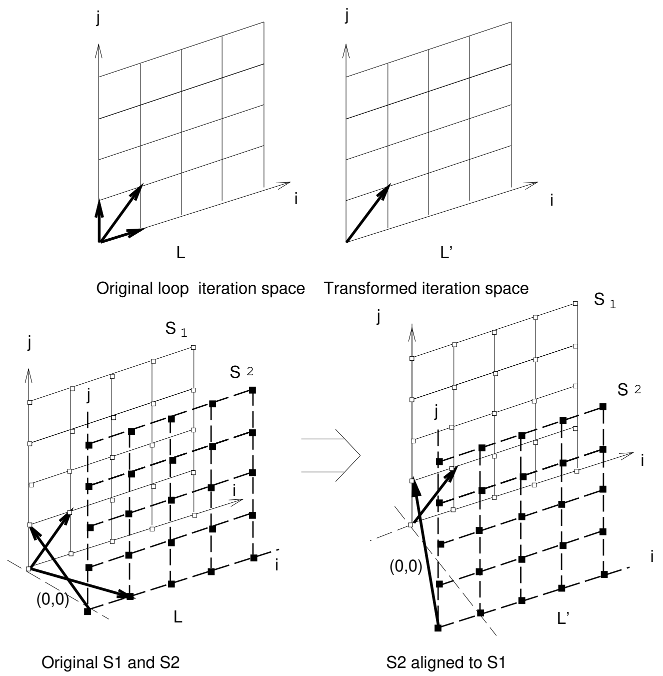


Figure 3: Aligning computations in a loop

The 2-dimensional mesh represents all computations of a given statement in the loop. The transformation shifts the computations of $\mathtt{S}_2$ to the left by one unit to form the transformed loop $\mathtt{L}'$. The (1,0) dependence becomes loop independent, and the other two dependences become (1,1).

The transformed loop is semantically equivalent to the original but has only one dependence carried by the loop, namely (1,1). The (0,1) dependence due to `B` also happens to become (1,1). This example clearly illustrates how Computational Alignment is capable of removing a "*false*" dependence that existed in the original program. It was capable of doing this because it was able to transform a portion of the loop body, namely statement $\mathtt{S}_2$, while keeping $\mathtt{S}_1$ unchanged. Traditional linear transformation can not eliminate dependences.

A transformation, such as the one in this example is useful because it allows, for example, a subsequent transformation to internalize [19, 20, 21] the (1,1) dependence to the inner loop in order to obtain parallelism that did not exist in the original organization of the statements, and could not have been obtained by traditional linear transformations alone.

To illustrate how Computational Alignment can be used for global optimization, consider the following sequence of loops:

```
L1 : for  i = 0, n
        for  j = 0, n
          A(i, j) = A(i, j - 1) + C
        end for
      end for

L2 : for  i = 0, n
        for  j = 0, n
          S1 :  A(i, j) = C(i - 1, j) + B(i - 1, j)
          S2 :  B(i, j) = A(i - 1, j - 1) + D
        end for
      end for
```

The first loop, L_1, has a (0,1) dependence, and outer loop parallelization would require a partitioning of array A by rows. On the other hand, L_2 has dependence (1,1) due to A and (1,0) due to B. If the same data partitioning would be used for loop L_2 as is used for L_1, then much communication would result. Since the references are of the form $i + c$, the access vectors are the same as the dependences; that is, the elements in A are accessed along the (0,1) direction in loop L_1 and along the (1,1) direction in loop L_2. Computational Alignment can align loop L_2 to L_1 so that A's access vector in L_2 becomes the same as that in L_1; i.e., the (1,1) dependence is transformed to (0,1). Ignoring again the changes to the bounds, the aligned program becomes:

```
L1 : for  i = 0, n
        for  j = 0, n
          A(i, j) = A(i, j - 1) + C
        end for
      end for

L2' : for  i = 0, n
        for  j = 0, n
          S1 :  A(i - 1, j) = C(i - 2, j) + B(i - 2, j)
          S2 :  B(i, j) = A(i - 1, j - 1) + D
        end for
      end for
```

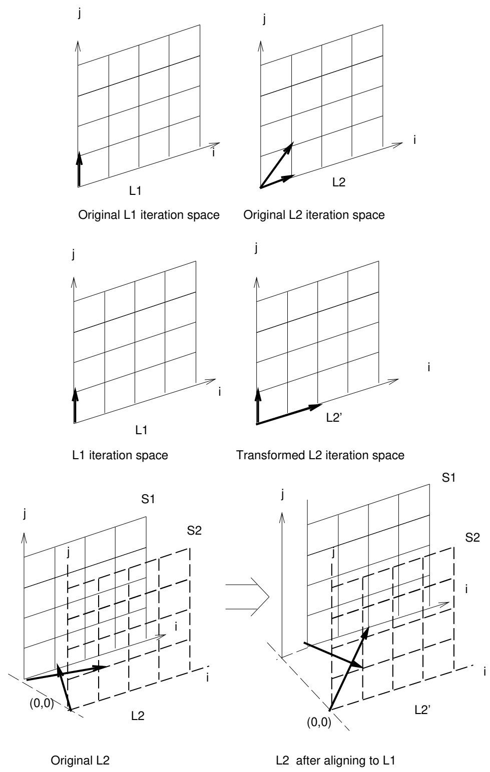


Figure 4: Aligning computations across loops

Figure 4 illustrates the transformation. The top part of the figure shows the original iteration spaces for $\mathtt{L}_1$ and $\mathtt{L}_2$; the transformed iteration spaces for the loops are shown in the middle. Note that $\mathtt{L}_1$ and $\mathtt{L}_2'$ are more "uniform" with respect to the dependences than $\mathtt{L}_1$ and $\mathtt{L}_2$. The bottom part of the figure shows the iteration spaces for statements $\mathtt{S}_1$ and $\mathtt{S}_2$ in $\mathtt{L}_2$ and $\mathtt{L}_2'$ individually as in the earlier example. The transformation involves shifting the iteration space of $\mathtt{S}_1$ to the right by one.

The transformed loop will continue to have communication due to accesses to `B`, but the accesses to array `A` have been modified to better suit row partitioning. In other words, `A`'s access vectors in loop $\mathtt{L}_2$ are transformed to make the two loops more uniform. Although the transformation in this particular case is a simple shift, it can be any linear function. The program model can also be much more general than the one considered in the example; for instance, $\mathtt{L}_1$ and $\mathtt{L}_2$ can have common nestings to form an imperfect loop.

3.2 Salient features of Computational Alignment

Because Computational Alignment fundamentally changes the dependence structure of a loop when aligning the computations of a portion of the body to other computations, it transforms only a subset of the dependences. This has three immediate implications:

1. Computational Alignment can be used to eliminate some of the "false" dependences in the loop. In the first example above, the (1,0) dependence between S_1 and S_2 was eliminated. Traditional linear loop transformations cannot eliminate dependences.

2. Computational Alignment effectively increases the search space for the optimal transformation by fundamentally changing the dependence structure and thus exposing an area of search space not previously available. In the first example, the original loop has three dependences, and there does not exist a communication free mapping of the iterations. The aligned loop, however, has a communication free mapping because it has only one dependence, namely (1,1).

3. Computational Alignment changes the references in the computations that are being aligned, thus changing the access vectors corresponding to the "false" dependences. Computational Alignment can therefore align the access vectors in a loop to those in some other loop. This makes Computational Alignment highly suitable for global optimizations. As noted in the earlier section, a traditional linear loop transformation can not change access vectors.

Not shown in the above examples is the fact that Computational Alignment can also be used to move computations relative to data, allowing Computational Alignment to be the *dual* of data alignment.

1. Computational Alignment can drastically reduce the number of ownership tests [31, 29, 30] required and enable a better computation partitioning. Just as data alignment moves the data relative to other data in order to make computations in the body execute on the same processor, Computational Alignment can move computations relative to other computations and achieve the same objective. Whenever a data alignment is not possible[1] then Computational Alignment can be used instead to achieve the same effect. We expect local optimization algorithms to employ both data and Computational Alignments to minimize (and possibly eliminate) ownership tests.

2. Computational Alignment can be used to match the structure of a loop better to a given set of data distributions. A mismatch between the loop structure and the distributions results in unnecessary data movements and one way to alleviate the problem is to transform the references so as to reflect the distributions [32]. Computational Alignment is better than the only existing technique capable of doing this, namely Access Normalization [32], because it can transform subsets of references in the aligned computations.

A final important feature of Computational Alignment is that it can be applied to computations in an imperfectly nested loop. In fact, the techniques developed in this paper can be used to extend the applicability of existing transformations to imperfect nests.

The salient features we briefly described here will become evident in the following sections.

[1] There are three reasons why data alignment may not be possible *i*) there may already exist a priori alignments; *ii*) the alignment functions are too complex; *iii*) the left hand sides are references to the same array.

```
for  I₁ = l₁, u₁
  for  I₂ = l₂(I₁), u₂(I₁)
    ...
      for  Iₖ = lₖ(I₁, .., Iₖ₋₁), uₖ(I₁, .., Iₖ₋₁)
        L₁ : ...
        L₂ : ...
        ⋮
      end for
    ...
  end for
end for
```

Figure 5: The program model

4 Computational Alignment

In this section, we formally define Computation Alignment. We start by defining the program model to which the transformations can be applied. We end the section by presenting techniques for deriving the structure (i.e., references and loop bounds) of the transformed program.

4.1 Program model

The development that follows is based on the program model shown in Figure 5. Each statement $\mathtt{L_i}$ can be either an assignment statement or a nested loop. The loop bounds are expressed by $\mathtt{l_i}$ and $\mathtt{u_i}$, which are affine functions of the enclosing iterators. The loops are normalized so that the step size is one. The arrays in the loop body are indexed by affine functions of the enclosing iterators. We assume that the program segment does not contain procedure calls. Up to Section 5, we assume that the given program does not specify any specific data alignments or data partitions.

We also assume that the dimension of all statements $\mathtt{L_i}$ are the same, namely $(n-k)$ for some $n \geq k$. This assumption simplifies the presentation of the techniques, but the techniques themselves do not depend on it. If $n = k$, then $\mathtt{L_i}$ denotes a simple statement. If $k = 0$, then $\mathtt{L_i}$ and $\mathtt{L_j}$ are loops in a program with no common nesting. If $k \neq 0$ and $n > k$, we have an imperfectly nested loop. This program model is general enough to represent a nested loop as well as simple complete programs.

The iteration vector $\mathtt{I_i}$ for loop $\mathtt{L_i}$ is $(I_1, ..., I_k, I^i_{k+1}, ..., I^i_n)$, where $I^i_{k+1}, ..., I^i_n$ are the iterators inside $\mathtt{L_i}$. The iteration vector specifies an integer point representing an iteration in the iteration space under consideration. The bounds of $I_1, ..., I_k, I^i_{k+1}, ..., I^i_n$ characterize the iteration space $\mathcal{I}_i$ of $\mathtt{L_i}$. When the context of the loop under consideration is clear, the iteration vector is denoted simply by $\mathtt{I} = (I_1, ..., I_n)$ and the corresponding iteration space by $\mathcal{I}$. Also, when the use of homogeneous coordinate system is clear, $\mathtt{I}$ denotes $(I_1, ..., I_n, 1)$. An array reference can be represented by an $m \times (n+1)$ reference matrix and the iteration vector.

4.2 Computations and their alignment

Given the above program model, the iteration space is traditionally represented by a k-dimensional polyhedron where each integer point denotes the body consisting of all the $\mathtt{L_i}$'s. Instead, we consider an iteration space for each $\mathtt{L_i}$ separately, where each integer point in that iteration space represents an $\mathtt{L_i}$ computation. These iteration spaces can each be also represented by a k-dimensional polyhedron. The basic idea of computational alignment is to shift and orient these polyhedrons with respect to each other for the purposes of program optimization. Each such shift or orientation corresponds to a linear transformation of the corresponding statement $\mathtt{L_i}$. Thus, Computational Alignment is equivalent to a "piecewise" linear transformation of the computations of a given iteration space.

Before formalizing the notion of alignment we define a computation as follows:

Definition 2 (Computation) *Given a loop body $S = \langle L_1, \ldots, L_m \rangle$ in an r-dimensional iteration space $\mathcal{I}_r$, then $c(i_1, ..., i_r; s)$ is a computation where $(i_1, ..., i_r) \in \mathcal{I}_r$ and s is any subsequence of S.* □

The subsequence s typically contains a sequence of consecutive statements in S. In many cases, s will contain only a single statement (where $s = \mathtt{L_i}$), or it will contain the entire loop body S, in which case $c(i_1, ..., i_r; s)$ denotes a complete iteration. (Note that $c(\phi, s)$ denotes a program fragment s without enclosing loops.)

Definition 3 (Set of computations) *Given a computation $c(i_1, ..., i_r; s)$ in an r-dimensional iteration space $\mathcal{I}_r$. The corresponding set of computations $C(\mathcal{I}_r, s)$ is defined as all instances of $c(i_1, ..., i_r; s)$ in $\mathcal{I}_r$*

$$C(\mathcal{I}_r, s) = \{c(i_1, ..., i_r; s) \mid (i_1, ..., i_r) \in \mathcal{I}_r \}$$

and the set is said to have dimension r. □

When the context is clear, a computation s can refer to the entire set of computations due to all the enclosing iterators. In Figure 5, $c1(j_1, ..., j_k; L_1)$ is a computation for some iteration $(j_1, ..., j_k)$. The corresponding set of computations is $C(\mathcal{J}_k, L_1)$, where $\mathcal{J}_k$ is the k-dimensional iteration space. When the context is clear, we just refer to $\mathtt{L_1}$. When $\mathtt{L_1}$ itself is an (n-k)-dimensional nested loop, and s is a statement in its body then $c(j_1, ..., j_k, j^1_{k+1}, ..., j^1_n; s)$ is a computation and $C(\mathcal{J}, s)$ is the corresponding set of computations. This abstraction allows us to address computations in an imperfect nesting.

It is evident that the above definitions characterize computations at various levels of granularity depending on the choice of s and the enclosing loops. The computation can be a statement (assignment or loop-statement), a complete body, or a set of some enclosed computations.

Definition 4 (Computational Alignment) *The alignment of a set of computations, C_2, of dimension r onto another set, C_1, of dimension $r' \geq r$ is a linear function $f_c : C_2 \rightarrow C_1$.* □

Computational Alignment maps the computations of one set onto another set, preferably in a way that improves the overall behaviour of the program (assuming the sets will be mapped to processors similarly). Although, the definition includes the alignment of a lower dimensional set to a higher one, for the sake of simplifying the presentation of the algorithms that follows we assume that the computation sets are of the same dimension. Because the mapping function f_c is linear, it can be specified by an $(r+1) \times (r+1)$ transformation matrix in homogenous coordinates. Denoting the iteration space of the $I_1, ..., I_k$ loops in Figure 5 by $\mathcal{I}_k$, the set of computations $C(\mathcal{I}_k, \mathtt{L_j})$ can be aligned to $C(\mathcal{I}_k, \mathtt{L_i})$ by a $(k+1) \times (k+1)$ transformation applied to $\mathtt{L_j}$. Similarly, the entire set of iterations inside $\mathtt{L_m}$ can be aligned to those inside $\mathtt{L_1}$ by a $(n+1) \times (n+1)$ transformation applied to $\mathtt{L_m}$'s iteration space.

An important special case of Computational Alignment is when a set of computations, $C(\mathcal{I}, S)$, with S being the entire loop body, is aligned to itself. These *self alignments* correspond to traditional linear loop transformations.[2] It is also important to note that Computational Alignment can be used not only to align perfectly nested computations, but also imperfectly nested computations.

Alignments do not in all cases result in transformed programs that are semantically equivalent to the original. We refer to those Computational Alignments that result in semantically equivalent programs as being *legal*.

Theorem 1 *A Computational Alignment, f_c, is legal iff*

1. *the sign of a dependence is the same in both the original program and the transformed program, and*

2. *there is a one to one correspondence between the set of computations in the original program and the transformed program.* □

Proof : Since the theorem is a slight modification of the one in linear transformations theory [17], we only provide an informal proof here. The first condition follows from the first principle of program transformation [2, 17]. It ensures that the new data dependence relations will lexicographically have the same sign as the original dependences so that they have the same temporal order, ensuring that correct values are computed.

The second condition ensures that the same computations are performed in the transformed program as in the original program - no more no less. This second condition is

[2] Linear loop transformations normally do not involve constant offsets which Computational Alignment allows.

implicit in linear loop transformations, since the transformations are applied to entire iterations, and the invertibility of the transformation and bound computations ensure that there is a one to one correspondence between iterations in the original program and the transformed program. In our case, the transformation may be applied to only parts of the loop body and thus the condition must be explicit. □

In the remainder of the section we show how to derive the structure of a transformed program after a Computational Alignment.

4.3 Deriving transformed loop structure

Suppose that a set of computations, C_2, in an n-dimensional loop is to be aligned to another set C_1 in the loop by the linear function f_c. We apply f_c to C_2; (C_1 is not transformed). The original and new iterators are given the same "names", but the new loop bounds must be computed for all iterators, including those common to C_1 and C_2. The references in C_2 must also be adjusted. In some cases, conditionals, called *guards*, must be introduced to enable only the appropriate computations in order to satisfy the second legality criterion.

The techniques developed in this paper for computing the new loop bounds form the basis for transforming imperfectly nested loops. Any linear transformation that is further applied to the nest could be combined with f_c, but for simplicity, we assume here that these additional transformations are applied only after the alignment.

Since C_2 and C_1 may be of dimension less than n, f_c may be of dimension less than $(n+1)$. The array references, however, can potentially be in terms of all n enclosing iterators, so the transformation can affect the bounds for all enclosing iterators common to C_2 and C_1. We therefore pad f_c to obtain an $(n+1) \times (n+1)$ transformation. Assuming that the r-dimensional computation sets C_2 and C_1 are defined by the enclosing iterators $(I_p, ..., I_{p+r-1})$, Algorithm CA-fill() of Figure 6, pads a given transformation f_c to construct a $(n+1) \times (n+1)$ transformation ${f_c}'$. The transformation f_c' maps all n iterators in the loop such that the iterators other than $(I_p, ..., I_{p+r-1})$ map onto themselves. For the remainder of this paper, we assume all alignment functions are properly padded.

New references: Suppose $\mathtt{L}_2$ is to be aligned to $\mathtt{L}_1$ by f_c. The "names" of the new iterators in both $\mathtt{L}_1$ and $\mathtt{L}_2$ will be the same as the old ones so that none of the references in $\mathtt{L}_1$ change. A reference in $\mathtt{L}_2$ represented by an $m \times (n+1)$ reference matrix R becomes a reference R' in the transformed loop:

$$R' = R \cdot f_c$$

New loop bounds: Suppose again that $\mathtt{L}_2$ is to be aligned to $\mathtt{L}_1$ by f_c. Since the transformation is applied to $\mathtt{L}_2$ only, the bounds of all the enclosing iterators of $\mathtt{L}_1$ do not change. Suppose β_1 and β_2 are the original bounds matrices for $\mathtt{L}_1$ and $\mathtt{L}_2$, respectively. The bounds for the common iterators, $I_1, ..., I_k$, are represented in both β_1 andβ_2. The new bounds matrix for $\mathtt{L}_2$, β_2' can be computed from the original bounds matrix β_2:

$$\beta_2' = \beta_2 \cdot f_c$$

Algorithm 1 : *CA-fill()*
```
input:  (r+1)-dimensional square matrix f_c, p, and r
output:  padded (n+1)-dimension square matrix f'_c
begin

  1. f'_c ← I, the (n+1)-dimensional identity matrix.
     /* nothing is transformed */

  2.   for i = p, p+r-1
         for j = p, p+r-1
           f'_c(i,j) ← f_c(i-p+1, j-p+1)
         end for
       end for

  3.   for i = p, p+r-1
         f'_c(i,n+1) ← f_c(i-p+1, r+1)
       end for
     /* I_p to I_{p+r-1} are transformed */

end
```

Figure 6: Padding an alignment

Then Extended Fourier-Motzkin elimination [27, 28, 26] can be applied to β_2' to obtain the new $\mathtt{L_2}$ loop bounds.

These new bounds are from the perspective of transformed $\mathtt{L_2}$. They can be used directly if there are no enclosing iterators common to both $\mathtt{L_1}$ and $\mathtt{L_2}$. In the general case, however, which occurs frequently in practice, there are enclosing iterators common to both $\mathtt{L_1}$ and $\mathtt{L_2}$. In this case, the original loop bound, which should apply to $\mathtt{L_1}$, and the transformed bounds, which should apply to $\mathtt{L_2}$, must be combined. One way to achieve this is to take the union of both the original and the transformed iteration space and then express the union in terms of a set of iterators. Algorithm CA-bounds() outlined in Figure 7, computes "conservative" new loop bounds.

Referring to Figure 7 , suppose that the computation set $C(\mathcal{I}_k, \mathtt{L_2})$ is being aligned to $C(\mathcal{I}_k, \mathtt{L_1})$ by a unimodular f_c, so that the transformed loop has unit strides (as in the original loop). β_1 and β_2 are the original bound matrices for $\mathtt{L_1}$ and $\mathtt{L_2}$ respectively. Step 3 gives the new bounds matrix as though only $\mathtt{L_2}$ is enclosed in $I_1, ..., I_k$. Step 4 provides the exact loop bounds from this new bounds matrix by applying Fourier-Motzkin variable elimination. Step 5 modifies the bounds obtained for $I_1, ..., I_k$ to account for the common nesting of $\mathtt{L_1}$ and $\mathtt{L_2}$.

This is done by computing the subsumption of the iteration spaces of $\mathtt{L}_1$ and the transformed $\mathtt{L}_2$. Because the union of the $\mathtt{L}_1$ and $\mathtt{L}_2$ polyhedra may not be convex, and because the above algorithm is conservative, the subsumption computed above may not be the exact union.

Conditional statements called *guards* which enable the appropriate computations and prevent the execution of other computations, must be introduced for two reasons. First, because the computed subsumption may not be exact, the new bounds may cause new and previously non-existing iterations to execute if they are not prevented. Second, an iteration in the union may correspond to only one of the two or both of the computations. For example, in Figure 8, iteration `p1` has no computations; `p2` has only $\mathtt{L}_2$ computation; `p3` has both $\mathtt{L}_1$ and $\mathtt{L}_2$ computations; and `p4` has only $\mathtt{L}_1$ computation.

Definition 5 (Guard) *Given a set of inequalities β, a guard for β, denoted $g(\beta)$, is a conditional statement that holds whenever the iterators satisfy the inequalities in β.* □

Aligning $\mathtt{L}_2$ to $\mathtt{L}_1$ in Figure 5 results in the need for a guard $g(\beta_1')$ for $\mathtt{L}_1$, where β_1' denotes the bounds for $I_1, ..., I_k$ in the original program. The guard $g(\beta_1')$ is a conditional that evaluates to true in iteration $(i_1, ..., i_k)$ iff the iterators satisfy all the inequalities in β_1'. The guard for $\mathtt{L}_2$ is $g(\beta_2')$, where β_2' are the $I_1, ..., I_k$ bounds from the transformed bounds matrix β_2'.

Example 4 Consider the following double loop with original bound matrices β_1 and β_2 for $\mathtt{L}_1$ and $\mathtt{L}_2$.

```
for  i = 0,n
  for  j = 0,n
    L1 :  A(i,j) = A(i - 1,j - 1)
    L2 :  B(i,j) = A(i - j,j)
  end for
end for
```

$$\beta_1 = \beta_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & n \\ 0 & -1 & n \end{bmatrix}$$

Suppose $\mathtt{L}_2$ is aligned to $\mathtt{L}_1$ with f_c

$$f_c = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Algorithm 2 : *CA-bounds():*

```
/* Computes bounds for transformed program */
input:  original program, f_c
output:  L'_i^j and U'_i^j new lower and upper bounds for 1 ≤ i ≤ n, 1 ≤ j ≤ 2
/* I_1,...,I_k, I_{k+1}^j,...,I_n^j are the iterators
for loop L_j for 1 ≤ j ≤ 2.  β_1 and β_2 original bounds matrices for L_1 and L_2.
β_1 and β_2 both contain bounds for the common iterators I_1,...,I_k.  */

begin
  1.  β_1 ← L_1 bounds matrix
  2.  β_2 ← L_2 bounds matrix
  3.  β_2 ← β_2 · f_c
  4.  /* Find new bounds due to f_c */
    for i = n, 1
      eliminate I_i from β_2
      β_2 ← new β_2 with I_i eliminated
      L'_i^2 ← max(lower bounds for I_i)
      U'_i^2 ← min(upper bounds for I_i)
      /* These are bounds for I_i due to transformed L_2 */
    end for
  5.  /*Find new bounds for the program */
    for i = 1,k
      L'_i ← min(l_i, L'_i^2)
      U'_i ← max(u_i, U'_i^2)
    end for
    for i = k+1, n
      L'_i^1 ← l_i^1
      U'_i^1 ← u_i^1
    end for
end
```

Figure 7: Computing new loop bounds

The new bounds matrix for $\mathtt{L_2}$ would then be:

$$\beta_2' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & n \\ 0 & -1 & n \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & n \\ 0 & -1 & n \end{bmatrix}$$

Applying Fourier-Motzkin variable elimination to β_2' results in the bounds:

$$-n \leq i \leq n \quad and \quad max(0, -i) \leq j \leq min(n, n-i)$$

Computing the union of the two sets of bounds as in CA-bounds() we obtain:

$$-n \leq i \leq n \quad and \quad min(0, max(0, -i)) \leq j \leq max(n, min(n, n-i))$$

which is

$$-n \leq i \leq n \quad and \quad 0 \leq j \leq n$$

When simplified, the guards for $\mathtt{L_1}$ and $\mathtt{L_2}$ are $g(\beta_1)$ and $g(\beta_2')$ respectively,

Figure 8 shows the loop before and after the alignment. CA-bounds() is computes a conservative union. In practice, the bounds can usually be made tighter. For example, modifying the lower bound of `j` from $min(0, max(0, -i))$ to $max(0, -i)$, steps off in this particular example, all of the iterations with no computations. The guards can also be simplified by removing redundant conditions. In this example, conditions $j \geq 0$, $i \leq n$, and $j \leq n$ are all redundant. These optimizations lead to the following transformed loop.

```
for  i = -n,n
  for  j = max(0,-i),n
    (i ≥ 0)              L1 :   A(i,j) = A(i-1,j-1)
    (j ≤ min(n,n-i))  L2 :   B(i+j,j) = A(i,j)
  end for
end for
```

□

The techniques discussed above are directly applicable to the transformation of imperfectly nested loops. Since the existing techniques assume a perfect nesting and most programs often contain imperfect loops, the techniques developed in this and later sections significantly advance the applicability of linear transformations to more general programs.

The run-time overhead due to empty iterations and guards can be considerable. In Section 7 we discuss an algorithm that provides tight loop bounds by computing the convex-hull [33] of the union of the aligned computation sets. This reduces the number of empty iterations to a minimum for perfectly nested transformed loop. We also present an algorithm capable of generating a *guard-free* transformed program for any arbitrary alignment and program.

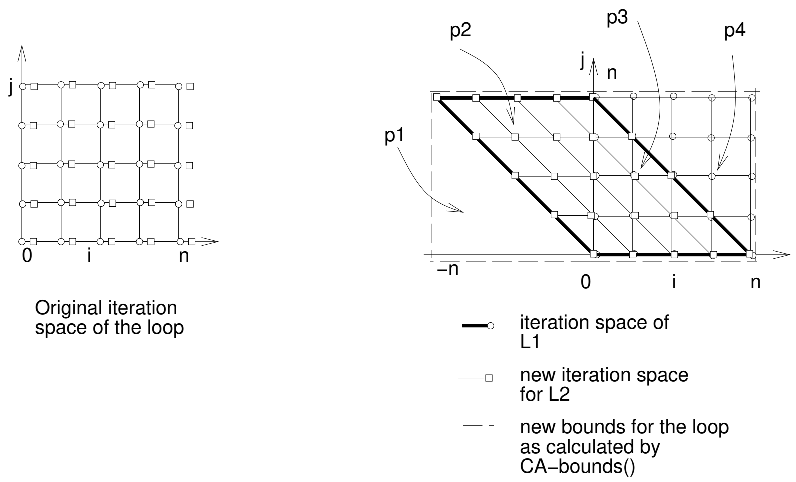


Figure 8: Deriving new loop bounds

5 Two simple alignments

The definition of Computational Alignment in Section 4 is very general in that it does not suggest which alignment function should be applied. The choice of alignment function will depend on the desired goal of the alignment. In this section, we discuss two subclasses of Computational Alignment which are simple and yet very useful. The first of these transformations is called Freeing Computational Alignment (FCA); it can free a loop of a dependence. The second is called Isomerizing Computational Alignment (ICA); it can align computations in a loop(s) to given data alignments and distributions.

5.1 Freeing Computational Alignment

FCA is a Computational Alignment that can transform one inter-statement loop carried dependence into another one. The name comes from the fact that it is often used to free the dependence entirely (by transforming it to a loop independent dependence).

Consider the program model in Figure 5 again. A dependence $\mathtt{d} = (d_1, ..., d_k)$ between $\mathtt{L_1}$ and $\mathtt{L_2}$, for example, is a precedence relation in the k dimensional iteration space. Since, each $\mathtt{L_i}$ can be an $(n-k)$ dimensional loop, we pad $\mathtt{d}$ with $(n-k)$ zeros to get a vector in n dimensions:

$$\mathtt{d} = (d_1, ..., d_k, 0, ..., 0)$$

Definition 6 (FCA) *A FCA for a dependence* $\mathtt{d} = \mathtt{L_1}\delta\mathtt{L_2}$ *is a function* f_c *that aligns* $\mathtt{L_2}$ *to* $\mathtt{L_1}$ *so that* $\mathtt{d}$ *is transformed to a given dependence* $\mathtt{d}'$. □

In many cases, $\mathtt{d}'$ is $\mathtt{0}$ so that the loop is "freed" of $\mathtt{d}$.

We now show how to derive a transformation f_c that is an FCA. Without loss of generality, consider a flow dependence $\mathtt{d} = \mathtt{L_1}\delta\mathtt{L_2}$. Let the LHS reference in $\mathtt{L_1}$ be represented by reference matrix A_w, and the RHS reference in $\mathtt{L_2}$ be represented by A_r. We can define U_1, U_2, $\mathtt{d_w}$ and $\mathtt{d_r}$ such that

$$A_w = \left[\begin{array}{c|c} U_1 \ {}_{n\times n} & \mathtt{d_w} \ {}_{n\times 1} \\ \hline 0 \ {}_{1\times n} & 1 \end{array}\right] \quad A_r = \left[\begin{array}{c|c} U_2 \ {}_{n\times n} & \mathtt{d_r} \ {}_{n\times 1} \\ \hline 0 \ {}_{1\times n} & 1 \end{array}\right]$$

The vectors $\mathtt{d_w}$ and $\mathtt{d_r}$ are the constant terms in the references and without loss of generality, we assume that $\mathtt{d_w}$ is $\mathtt{0}$. If a given array element is accessed by $\mathtt{L_1}$ in iteration $\mathtt{I}$ and by $\mathtt{L_2}$ in iteration $\mathtt{I}'$, then the dependence $\mathtt{d}$ is $\mathtt{I}' - \mathtt{I}$. This dependence is characterized by the following equation.

$$U_1\mathtt{I} = U_2\mathtt{I}' + \mathtt{d_r}$$

Using a matrix-offset notation, suppose the alignment function $f_c = (T\ {}_{n\times n}$ and offset $\mathtt{t}\ {}_{n\times 1})$ transforms dependence $\mathtt{d}$ into another $\mathtt{d}'$. Then we have:

$$U_1\mathtt{I} = U_2.T\mathtt{I}' + \mathtt{d_r} + \mathtt{t} \tag{1}$$

Solving for $\mathtt{d}'$, which is defined by $\mathtt{I}' - \mathtt{I}$, we obtain f_c:

$$f_c : T = U_2^{-1}.U_1 \text{ and offset } \mathtt{t} = -U_1\mathtt{d}' - \mathtt{d_r} \tag{2}$$

Thus, in homogeneous coordinates, the particular FCA that transforms dependence $\mathtt{d}$ to $\mathtt{0}$ can be obtained from Equation 2 by substituting $\mathtt{0}$ for $\mathtt{d}'$.

$$f_c = \left[\begin{array}{c|c} U_2^{-1}.U_1 \ {}_{n\times n} & -\mathtt{d_r} \ {}_{n\times 1} \\ \hline 0 \ {}_{1\times n} & 1 \end{array}\right] \tag{3}$$

Note that there is an implicit assumption in the above development, namely that the reference matrix is invertible. We are currently extending the technique to handle singular reference matrices.

References of the form $i \pm c$ for some constant c in each dimension are common. The dependence $\mathtt{d}$ to be freed is then $-\mathtt{d_r}$, and the FCA is just a shift, i.e.,

$$f_c = \left[\begin{array}{c|c} \mathtt{I} \ {}_{n\times n} & \mathtt{d} \ {}_{n\times 1} \\ \hline 0 \ {}_{1\times n} & 1 \end{array}\right]$$

The textual order of $\mathtt{L_i}$ determines the sign of the loop independent dependences. Freeing $\mathtt{d} = \mathtt{L_i}\ \delta \mathtt{L_j}$, when $i < j$ is always possible. On the other hand when $i > j$, the freeing of $\mathtt{d}$ must be followed by a valid textual interchange of $\mathtt{L_i}$ and $\mathtt{L_j}$. The dependences where $i = j$ are not affected if the FCA only shifts a computation (that is, when T is an identity matrix). FCA never affects the dependences in $\mathtt{L_i}$, but other dependences in $\mathtt{L_j}$ can be affected, especially when the applied FCA is a rotation of the iteration space. Also, dependences between $\mathtt{L_i}$ and $\mathtt{L_j}$ beside the one being freed could change. If there are dependences between $\mathtt{L_j}$ and any other statement, then they can be affected as well. (In short, if $\mathtt{L_j}$ is transformed by an FCA, then all dependences to and from $\mathtt{L_j}$ in a dependence graph can be affected.)

The loop bounds of an FCA transformed structure can be determined as discussed in Sections 4 and 7.

Example 5 An FCA can align $\mathtt{S_2}$ to $\mathtt{S_1}$ in the following double loop in order to free the (0,1) dependence with respect to B:

```
for  i = 1,n
  for  j = 1,n
    S1 :  A(i,j) = A(i − 1,j − 1) + B(i,j − 1)
    S2 :  B(i,j) = A(i − 1,j)
  end for
end for
```

to obtain the following transformed loop:

```
for  i = 1,n
  for  j = 0,n
    (j ≥ 1)  S1 :  B(i,j) = A(i − 1,j)
    (j ≤ n − 1)  S2 :  A(i,j + 1) = A(i − 1,j) + B(i,j)
  end for
end for
```

Note that in eliminating the (0,1) dependence, the (1,0) dependence with respect to array A is transformed into a (1,1) dependence.

5.2 Isomerizing Computational Alignment

ICA is a Computational Alignment that makes two array references similar. Suppose A_{w1} and A_{w2}, are two LHS references to an array, and we already know how the loop will be partitioned. To derive a unique, communication free data partitioning, the two references, A_{w1} and A_{w2} should be made the same. This can be done with an ICA.

As another example, suppose that A_w and B_w are LHS references to arrays A and B and that their data distribution is the same (possibly because of the prior alignments). A computation rule such as the ownership rule can be taken out of the loop and efficient SPMD code can be produced if the statements could be aligned such that they evaluate to the same owners. The need for this type of optimization occurs frequently in practice, and ICA is an effective technique that can be applied towards this goal.

Definition 7 (ICA) *An ICA for a reference A_2 in* $\mathtt{L}_2$ *is a function f_c that aligns* $\mathtt{L}_2$ *to* $\mathtt{L}_1$ *so that*

$$A_2.f_c = A_1 \cdot \alpha$$

where A_1 is a reference in $\mathtt{L}_1$*, and α is a matrix that represents some relation that should hold between* $\mathtt{L}_1$ *and aligned* $\mathtt{L}_2$*.* □

A special case is when α is the identity matrix, in which case f_c makes the two LHS's references the same. ICA can match the loop to a distribution if the references A_1 and A_2 are to the the same array, or to different arrays with same distribution. When A_1 and A_2 are references of different arrays, α is used to represent the data alignment that maps an element in one array to another. In addition, ICA can further be generalized to cases where A_1 and A_2 are references to different arrays of the same dimension but with different distributions, represented by functions D_1 and D_2.[3] The alignment f_c is then a solution to

$$D_2 \cdot A_2 \cdot f_c = D_1 \cdot A_1 \cdot \alpha$$

Example 6 Consider the following loop, selected from Lapack:

```
for k = 0,n
  for j = 0,n
    S1 :   H(k,j) = H(k,j) − sum ∗ T1
    S2 :   H(k − 1,j) = H(k − 1,j) − sum ∗ T2
  end  for
end  for
```

$\mathtt{S}_2$ can be aligned to $\mathtt{S}_1$ with a transformation f_c that simply shifts the $\mathtt{S}_1$ computations to the right along the k axis to produce the following loop that has a unique LHS reference pattern. Irrespective of the way H is distributed, both of the LHS references evaluate to the same owner, making it possible to eliminate the ownership tests and thus avoiding unnecessary computations and data movements. The transformed loop is in fact fully parallel:

```
for k = 0,n + 1
  for j = 0,n
    S1 :   (k ≥ 1) :  H(k − 1,j) = H(k − 1,j) − sum ∗ T1
    S2 :   (k ≤ n) :  H(k − 1,j) = H(k − 1,j) − sum ∗ T2
  end  for
end  for
```

[3] For an r-dimensional processor array, and m-dimensional arrays, D_1 and D_2 are $k \times m$ linear functions mapping an array element to a processor.

Example 7 Consider the following loop that references two different arrays. Assume that A and B have been previously aligned such that A(i,j) and B(i-1,j) are collocated. (α in this case is a simple linear shift.) Since they are aligned, they will be distributed the same way.

```
for  i = 0,n
  for  j = 0,n
    S1 :  A(i,j) = ...
    S2 :  B(i,j+i) = ...
  end for
end for
```

The following ICA aligns the computations so that a loop partitioning can be accomplished without any ownership tests or data movements (but at the cost of introducing guards). Section 7 will describe how the guards can be eliminated.

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

The aligned loop becomes:

```
for  i = 0,n+1
  for  j = i,n+i
    (i ≤ n, j ≤ n) :  S1 :  A(i,j) = ...
    (i ≥ 1, j ≥ i) :  S2 :  B(i-1,j) = ...
  end for
end for
```

6 Compound FCAs and ICAs

A local and global optimization algorithm will typically need to employ a sequence of FCAs and ICAs to each loop nest. Comprehensive optimization algorithms are beyond the scope of this paper. However, we provide some hints as to how to construct optimizers of this type in this section.

In the context of local optimization, Comutational Alignment is useful both for *process-oriented* optimization, which transforms a given loop to map computations to processors and then to derive appropriate data distributions, and for *data-oriented* optimization which maps computations onto processors given specific data alignments and distributions. For the former case, we expect the optimizer to derive a sequence of FCAs and then apply this sequence. (This may be followed by additional traditional linear transformation.[4]) For the later case, the optimizer derives a sequence of ICAs in order to optimally parition the computations.

The set of uniform dependences of a loop, D, consists of the subset, D_f, of dependences which are between statements, and the subset $D_c = D - D_f$ of self dependences. A uniform dependence between `Li` and `Lj` can be eliminated or transformed to some other dependence by a legal FCA that simply shifts `Lj`. A dependence inside `Lj` can be made the same as a dependence inside `Li` by applying a legal FCA that rotates `Lj` with respect to `Li`.

Given a loop, the objective is to derive an FCA for each of the statements, so that if they are all applied, we obtain a transformed loop with a minimal number of dependences.[5] We achieve this objective by defining a set of transformations for each statement independently and then exhaustively searching for the best combination.

The set of alignments we consider for statement `Li` will contain shifting FCAs and rotating FCAs:

$$FCA_{L_i} = SFCA_{L_i} \bigcup RFCA_{L_i}$$

The set of $SFCA_{L_i}$ is the union over all dependences `d` related to the statement of each shifting FCA capable of eliminating `d` or changing it to another statement `d'`.

$$SFCA_{L_i} = \bigcup_{d \in D_f} SFCA_{L_i}(d) \bigcup \bigcup_{d \in D_f} \bigcup_{d' \in D} SFCA_{L_i}(d, d')$$

It should be noted that each FCA $SFCA_{L_i}(d)$ or $SFCA_{L_i}(d, d')$ will be equivalent to a sequence of one of the basic shifting FCAs:

FCA((0,0)) : no shift,

FCA((1,0)) : a unit shift along the `I1` dimension,

FCA((0,1)) : a unit shift along the `I2` dimension.

The set of $RFCA_{L_i}$ is the union over all the dependences of each rotating FCA capable of changing it to another dependence `d'`.

$$RFCA_{L_i} = \bigcup_{d \in D} \bigcup_{d' \in D} RFCA_{L_i}(d, d')$$

It should be noted that each of $SFCA_{L_i}$ and $RFCA_{L_i}$ is finite, and hence so is FCA_{L_i}. Because it is finite, it is possible to exhaustively search through all combinations to find the one that results in the minimal number of dependences.

Finding a sequence of ICAs on the other hand is easier. The idea is to consider the given data alignments and distributions, and to apply an ICA to each of the statements so that as

[4] As pointed out earlier, this can be integrated with each of the FCAs in the sequece.

[5] This criteria can be further refined; for instance the types of desired dependences.

many as possible have their left hand side evaluate to the same owner. SInce some of the ICAs may not be legal, it may not be possible to eliminate the ownerships test entirely. Further optimization is still possible by applying ICAs so that the statements execute in neighbouring processors.

7 Optimizing transformed code

Computational Alignment extends the linear loop transformations in a new dimension by being able to transform portions of a loop body independently. The examples, identified a variety of situations where Computational Alignment is useful. However, this additional power comes with an associated cost; in its basic form, an aligned program will usually not have precise loop bounds and will require guards to step off empty iterations and to enable the appropriate computations in an iteration. These guards can entail a considerable amount of run-time overhead. In this section, we present two algorithms that improve the techniques discussed so far. The first of these, reduces the number of empty iterations by deriving tight bounds, while keeping the iteration space convex. The second eliminates empty iterations and guards altogether, but at the potential cost of introducing imperfect loops.

7.1 Bound Optimizations

The basic idea in computing bounds is to find the union of two convex polyhedrons, corresponding to the iteration spaces of the two computation sets involved. The (exact) union of two convex sets can be non-convex, yet a perfectly nested loop can unfortunately only characterize a convex iteration space. In order to keep the loop perfectly nested, we need to compute a convex set that subsumes the union. Making the convex set as tight as possible reduces overhead due to extra iterations and the guards they contain. It also helps to keep the load more balanced and allows for better prediction of available parallelism and communication requirements.

Algorithm CA-bounds() presented in Section 4 is simple and intuitive but does not derive a tight subsumption of the union in all cases. In this section, we present a method to compute tight bounds for any given original loop and unimodular alignment functions (when both the original and transformed loops have unit strides.) The computed bounds are optimal in that they result in a minimum number of empty iterations when the iteration space of the transformed loop is required to be convex.

The basic idea in Algorithm CA-bounds-optimal() of Figure 9 is to find the convex hull [33] of the union of the aligned computations. From the definition of the convex hull, the bounds provided by the description of this convex hull are tight. In fact, when the union is a convex polyhedron, this algorithm gives exact bounds.[6] E_1 and E_2 are the extreme points of iteration spaces $\mathcal{I}_1$ and $\mathcal{I}'_2$ respectively. Step 3 is based on the observation that an extreme point of the union is an extreme point of either one of the two iteration spaces. Step 3 simply takes the convex hull of the union of E_1 and E_2. This can be accomplished by applying

[6] The algorithm uses iteration spaces directly instead of bounds matrices.

Algorithm 3 : *CA-bounds-optimal()*
/* *Computes optimal bounds after aligning* `L2` *to* `L1` *by* f_c */
/* $\mathcal{I}_1$ *and* $\mathcal{I}'_2$ *iteration spaces of* `L1` *and* `L'2` *the transformed* `L2` */
/* $\mathcal{I}_1$ *and* $\mathcal{I}'_2$ *are defined by* β_1 *and* β'_2,
the bounds matrices of `L1` *and* `L'2` */

```
input:  I1, I'2
output:  new bounds matrix β'
begin
  1.  E1 ← extreme points of I1
  2.  E2 ← extreme points of I'2
  3.  H ← CH(E1 U E2)
  4.  u ← any point such that u ∈ I1 or u ∈ I'2
  5.  β ← φ
  6.  for each bounding hyperplane (h(I) = 0) ∈ H
        if h(u) ≤ 0 then β ← β ∪ (h(I) ≤ 0)
        else β ← β ∪ (h(I) ≥ 0)
      end for
  7.  β' ← FM(β)
end
```

Figure 9: Optimal transformed bounds

the *gift-wrapping* or the *beneath-beyond method* [33]. The convex hull, i.e. set H, is the set of bounding hyperplanes of the form $h(\mathtt{I}) = 0$. A given $h(\mathtt{I})$ is either a lower bound or an upper bound depending on which side of the hyperplane an arbitrary point in the union lies. Step 4 chooses some point u known to be in the union. In Step 6, we substitute u in each of the bounding hyperplanes in H. If $h(u) \leq 0$ then $h(\mathtt{I})$ is a lower bound and the inequality $h(\mathtt{I}) \leq 0$ is added to the set β; otherwise it is an upper bound in which case $h(\mathtt{I}) \geq 0$ is added to the set β. Finally, we apply Fourier-Motzkin variable elimination in Step 7 to obtain the new loop bounds β'.

Example 8 Consider the program of Example 4. The extreme points of $\mathcal{I}_1$ are (0,0), (n,0), (0,n) and (n,n). Those for $\mathcal{I}'_2$ are (0,0), (n,0), (-n,n) and (0,n). Step 3 in CA-bounds-optimal() provides the equalities, from which Step 6 produces the following inequalities:

$$i \leq n, \quad j \geq 0, \quad j \geq -i, \quad j \leq n$$

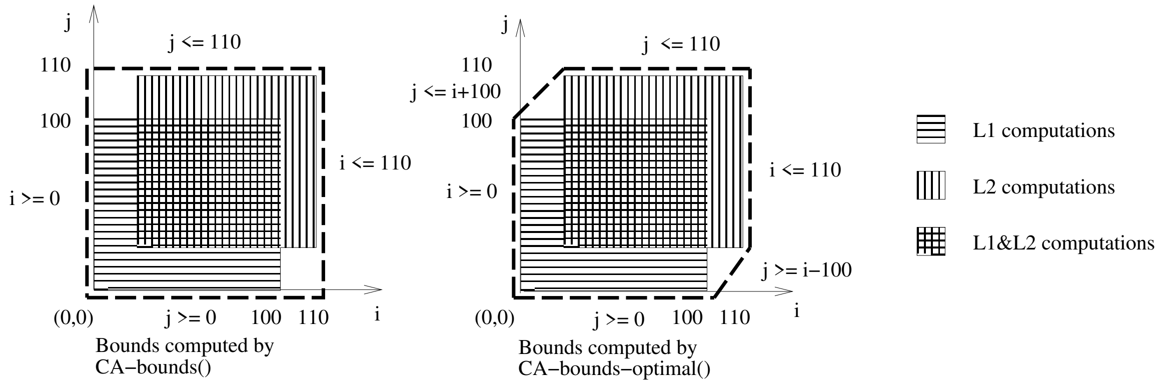


Figure 10: New loop bounds by CA-bounds-optimal()

After variable elimination, they provide the bounds,

$$-n \leq i \leq n, \quad max(0, -i) \leq j \leq n$$

The loop bounds happen to be exact in this case, since the union is a convex polyhedron. □

Figure 10 gives another example, where the union is a non-convex polygon. The left side shows the loop bounds that are derived by CA-bounds(), and the right side of the figure show the bounds obtained by using CA-bounds-optimal.

The algorithm CA-bounds-optimal() is also useful in the context of computing the *Data Access Descriptors* [34] as summaries of the data access in a loop or a procedure. It is also useful in other cases that require the computation of unions such as array privatization [35, 36]. In both cases, our algorithm provides tighter descriptions than the existing algorithms.

7.2 Guard Optimizations

The algorithm CA-bounds-optimal() discussed above produces the tight convex loop bounds for the aligned program. Although this reduces the number of empty iterations, guards are still required for two reasons. First, it is necessary to step off the empty iterations which are left. Second, some of the non-empty iterations contain only one of the computations, and guards are needed to block the other computation. The guards lead to extra execution overhead and in addition could prevent some later optimizations. A simple first step would be to remove the redundant conditions from the guards. We did this for instance, in Example 4. A better alternative, however, is to eliminate the guards altogether. We present here a method to generate transformed code that is *guard-free*.

The premise for generating the convex hull of the possibly concave union of iteration spaces is that a perfect nest can only express a convex set. However, we will show that *any concave polyhedron can be expressed by an imperfect loop*, so guard-free code can be generated if imperfect nests are acceptable. If it is important to keep the transformed loop perfect in order to allow future transformations, one has to derive tight bounds by CA-bounds-optimal() and keep the necessary guards. However, if the overhead due to guards is unacceptable, one can express the exact bounds in a guard-less imperfect loop, possibly restricting the future transformations to only those that apply to imperfect loop nests. In the following presentation we assume that the original code contains only compile time constants in the bound expressions.[7]

The basic idea is as follows. The points in the union of the iteration spaces of computations $\mathtt{L}_1$ and $\mathtt{L}_2$ can be classified into three categories, namely, those i) with only an $\mathtt{L}_1$ computation, ii) with only an $\mathtt{L}_2$ computation, and iii) with both the computations.[8] The idea is to partition the space into segments so that all points in a segment belong to the same category. Each of these segments can be specified by a range along the $\mathtt{I}_1$ axis, and by beginning and ending functions in $\mathtt{I}_1$ that delineate the segment with respect to the $\mathtt{I}_2$ axis. These functions do not have to be monotonous. Each segment defined in an $\mathtt{I}_1$ range spans the entire range. We generate code to execute iterations in these segments in correct lexicographic order.

Figure 11 helps illustrate the idea further. It shows the iteration spaces of $\mathtt{L}_1$ and $\mathtt{L}_2$, the union of which is concave. The integers 0, i_0 - i_3 define $\mathtt{I}_1$ ranges for this example. Linear functions in $\mathtt{I}_1$, h_0 - h_4, denote lines. For the range $[0, i_0 - 1]$, the lower bound of the segment is given by 0, and the upper bound by h_1; this segment has only $\mathtt{L}_1$ computations. For the range $[i_0, i_1]$, three segments must be defined. The first is delineated by 0 and h_0 respectively, and has only $\mathtt{L}_1$ computations. The second is delineated by h_0 and $min(h_1, h_3)$, and has both $\mathtt{L}_1$ and $\mathtt{L}_2$ computations. The third segment is delineated by $min(h_1, h_3)+1$ and $min(h_2, h_4)$, and has only $\mathtt{L}_2$ computations.

Given these segments, we can construct a program with a sequence of four loops, one for each defined $\mathtt{I}_1$ range. Each of these loops consists of a sequence of inner $\mathtt{I}_2$ loops, one for each segment defined in that particular $\mathtt{I}_1$ range. The individual bound expressions are chosen so as to delineate the segment. For example, Figure 12 shows the code generated for the three segments defined in the range $[i_0, i_1]$.

Although the above ideas extend to any dimension, for ease of illustration we present a two dimensional version of the algorithm gen-guard-free-code() in Figure 13. It assumes the aligned computations have two commonly enclosing iterators, and that $\mathtt{L}_2$ is transformed to $\mathtt{L}'_2$ by aligning $\mathtt{L}_2$ to $\mathtt{L}_1$ by f_c. $\mathtt{L}_1$ and $\mathtt{L}_2$ have the enclosing common iterators $\mathtt{I}_1$ and $\mathtt{I}_2$. $\mathcal{I}_1$ is the iteration space of $\mathtt{L}_1$ the constants l and u are the lower and upper bounds of $\mathtt{I}_1$, and the affine functions $f(\mathtt{I}_1)$ and $g(\mathtt{I}_1)$ being the lower and upper bounds of $\mathtt{I}_2$. Let $\mathcal{I}'_2$ be the iteration space for $\mathtt{L}'_2$ with its corresponding bounds l', u', $f'(\mathtt{I}_1)$ and $g'(\mathtt{I}_1)$.

We first calculate the $\mathtt{I}_1$ ranges. Crossover() is a function that computes the points at which the bounds of $\mathtt{L}_1$ and $\mathtt{L}'_2$ intersect in the union and return projections on $\mathtt{I}_1$ axis. (When the intersection is not at an integer point, we take the floor of the values.) X_1 is the set of $\mathtt{I}_1$ bounds in $\mathtt{L}_1$ and $\mathtt{L}_2$ and the crossover points. These points define the $\mathtt{I}_1$ ranges. $Ranges1D()$ sorts the points and defines the $\mathtt{I}_1$ ranges accordingly. These ranges are closed (i.e., both end points belong to the range), and each range is associated with a label to indicate whether it

[7]This assumption stems from the fact that we are not taking the aid of a tool for symbolic comparisons.

[8]Since we are considering the actual union, there will not be a point with no computation.

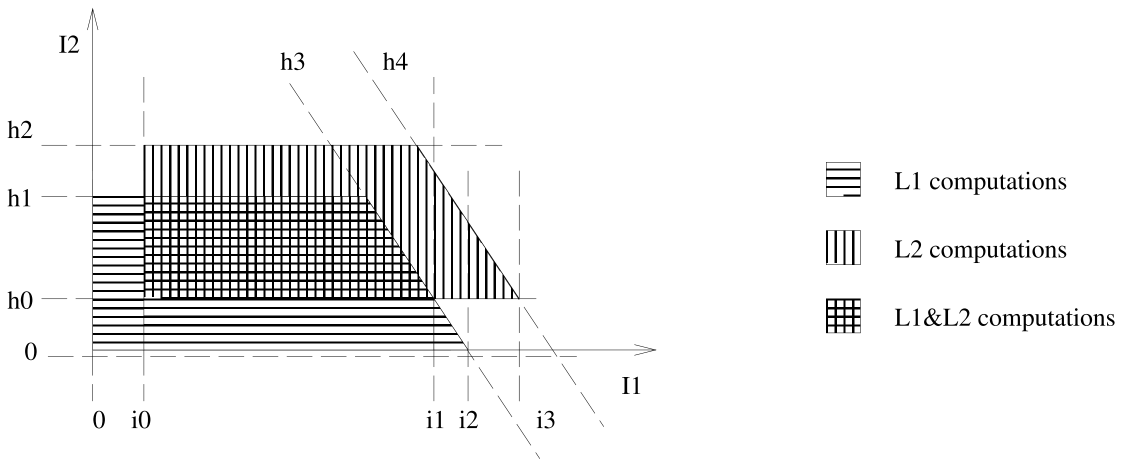


Figure 11: Segmenting the union

contains $\mathtt{L}_1$ computations, $\mathtt{L}_2$ computations or both. This is done by comparing a range with the $\mathtt{I}_1$ bounds of $\mathtt{L}_1$ and $\mathtt{L}_2$. $Comp()$ returns the label of a range.

Step 3 iterates over all computed $\mathtt{I}_1$ ranges, from the left to right. Nextrange() selects the next range to generate code for in each iteration. If the selected range $[i_l, i_r]$ has a label $\mathtt{L}_1$ or $\mathtt{L}_2$, a double loop corresponding to that range is generated (Steps 14 and 15). If the $[i_l, i_r]$ has a label $\mathtt{L}_1$&$\mathtt{L}_2$, we first generate a `for` statement for the $\mathtt{I}_1$ range $[i_l, i_r]$. We then partition $\mathtt{I}_2$ into ranges defining polygons that have only one or both of the computations. Because $[i_l, i_r]$ does not contain any intersection points, there exists a unique sequence of such polygons. These polygons are again labeled to identify the computation(s) they contain. We determine the order of these computations by substituting an arbitrary point u from $[i_l, i_r]$ in $f(\mathtt{I}_1)$, $f'(\mathtt{I}_1)$, $g(\mathtt{I}_1)$ and $g'(\mathtt{I}_1)$ to obtain X_2, and then sort X_2. $Ranges2D()$ takes these sorted points and defines $\mathtt{I}_2$ ranges. Each range consists of a pair of functions from the $\mathtt{L}_1$ and $\mathtt{L}_2'$ bounds that bound the range from below and above. An $\mathtt{I}_2$ loop is generated for each of these ranges from the bottom to the top with the corresponding computation(s) (Steps 9-13).

Example 9 Consider the alignment of Example 4 with `n` of 100. The crossover occurs at `i` =0, and 100. If we include the `i` bounds, the ranges for $X_1 = \{-100, 0, 100\}$ would be $R_1 = \langle[-100, -1], [0, 100]\rangle$, and the computation they are associated with would be $\mathtt{L}_2$ and $\mathtt{L}_1$&$\mathtt{L}_2$ respectively.

The the range $[-100, -1]$ contains only $\mathtt{L}_2$. We therefore, generate a double loop containing $\mathtt{L}_2$ (Step 15). Iterator `i` varies from -100 to -1 (with a step of 1). The `j` bounds are used as given.

The second range, $[0, 100]$, has both $\mathtt{L}_1$ and $\mathtt{L}_2$ computations. We generate a `for` statement for `i` first (Step 5). The `j` bounds are 0, 100, and $max(0, -i)$ and $min(100 - i, 100)$. We know that these bounds will not crossover within the range $[0, 100]$. Substituting, say, `i` =1 into these bound equations to sort them, we obtain the ranges $R_2 = \langle[0, min(100 -$

```
for  I1 = i0, i1
  for  I2 = 0, h0 − 1
    L1 :   ...
  end for
  for  I2 = h0, min(h1, h3)
    L1 :   ...
    L2 :   ...
  end for
  for  I2 = min(h1, h3) + 1, min(h2, h4)
    L2 :   ...
  end for
end for
```

Figure 12: Guard-free code template

$i, 100], [min(100 - i, 100] + 1, 100]\rangle$ associated with computations $\mathtt{L_1}\&\mathtt{L_2}$ and $\mathtt{L_1}$ respectively (Steps 6-8). (The ranges can further be simplified to $R_2 = \langle[0, 100 - i], [100 - i + 1, 100]\rangle$ knowing that i is positive.) We generate one `j` loop containing both computations (Step 11) followed by another `j` loop with only $\mathtt{L_1}$ (Step 12).
The final code thus becomes:

```
for  i = −100, −1
  for  j = max(0, −i), min(100 − i, 100)
    L2 :   B(i + j, j) = A(i, j)
  end for
end for
for  i = 0, 100
  for  j = 0, min(100 − i, 100)
    L1 :   A(i, j) = A(i − 1, j − 1)
    L2 :   B(i + j, j) = A(i, j)
  end for
  for  j = min(100 − i, 100) + 1, 100
    L1 :   A(i, j) = A(i − 1, j − 1)
  end for
end for
```

Algorithm 4 *: gen-guard-free-code()*
/ $\mathcal{I}_1$ and $\mathcal{I}'_2$ iteration spaces of* $\mathtt{L_1}$ *and* $\mathtt{L'_2}$ *the transformed* $\mathtt{L_2}$ **/*

```
input:  $\mathcal{I}_1$, $\mathcal{I}'_2$
output:  Guard-free program
begin
1.    $X_1 \leftarrow \{l, u, l', u'\} \cup crossover(\mathcal{I}_1, \mathcal{I}'_2)$
2.    $R_1 \leftarrow ranges1D(X_1)$
3.    while (there is a range in $R_1$) do
4.          $[i_l, i_r] \leftarrow nextrange(R_1)$
5.          if $comp([i_l, i_r])$ = "L$_1$&L$_2$" then
               gen(for $\mathtt{I_1 = i_l, i_r}$)
6.             $u \leftarrow$ arbitrary point $\in (i_l, i_r)$
7.             $X_2 \leftarrow \{f(u), g(u), f'(u), g'(u)\}$
8.             $R_2 \leftarrow ranges2D(X_2)$
9.             while (there is a range in $R_2$) do
10.                  $[j_l, j_r] \leftarrow nextrange(R_2)$
11.                  if $comp([j_l, j_r])$ = "L$_1$&L$_2$" then
                         gen(for $\mathtt{I_2 = j_l, j_r}$ / L$_1$ / L$_2$ / end  for)
12.                  elsif $comp([j_l, j_r])$ = "L$_1$" then
                         gen(for $\mathtt{I_2 = j_l, j_r}$ / L$_1$ / end  for)
13.                  else
                         gen(for $\mathtt{I_2 = j_l, j_r}$ / L$_2$ / end  for)
                     end if
               end while
14.         elsif $comp([j_l, j_r])$ = "L$_1$" then
               gen(for $\mathtt{I_1 = i_l, i_r}$ / for $\mathtt{I_2 = f(I_1), g(I_1)}$ / L$_1$ / end  for / end  for)
15.         else
               gen(for $\mathtt{I_1 = i_l, i_r}$ / for $\mathtt{I_2 = f'(I_1), g'(I_1)}$ / L$_2$ / end  for / end  for)
            end if
      end while
end
```

Figure 13: Generating guard-free transformed code

8 Concluding Remarks

In this paper, we have introduced and formalized Computational Alignment, a new class of program transformations that not only extends the existing linear loop transformations and data alignments in a new dimension but also provides a bridge to global transformations. Computational Alignment is significantly more powerful than existing loop and data transformations: *i*) it can transform subsets of dependences and references, and can potentially be used to eliminate inter-statement dependences and to better match loop structure to given data alignments and distributions; *ii*) because of this, it is sensitive to the location of data in that it can move the computation relative to data; *iii*) it applies to imperfect loop nests, and hence the techniques presented can be used to extend existing transformations; *iv*) it is the first loop transformation that can change access vectors, and *v*) it is the dual of data alignment, and can therefore potentially satisfy data alignment requirements without realigning the data. Linear transformations are just a special case of Computational Alignment.

Computational Alignment is highly suitable for global optimization because with its capabilities it can transform given loops to access data in similar ways.

We have also presented two important subclasses of Computational Alignment, namely FCA and ICA. A set of FCAs and ICAs will typically be applied in sequence to transform dependences so as to obtain loop transformations that are near optimal. These subclasses play an important role in global optimization as well. Other contributions of the paper include an algorithm to compute tight loop bounds which can be used to obtain more accurate data access descriptors [34] and to improve techniques for inter-procedural analysis and array privatization [35].

The flexibility and power of Computational Alignment comes, however, with additional execution overhead for empty iterations and guards when perfect nests are desired. We presented techniques that compute tight bounds and generate guard-free code for arbitrary alignment functions, but causing imperfect nesting in the process.

In related work, Allen et al. discuss loop alignment, a transformation that is similar to FCA in eliminating dependences [12]. However the scope of their work is limited to obtaining independent partitions in a single loop, and does not have the algebraic framework provided with Computational Alignment. Access Normalization [37] is similar to ICA with α set to the identity matrix, and treating the entire loop body uniformly. The subclass ICA is more effective than Access Normalization, because it can selectively transform a subset of the references instead of all of them, and because it can handle both alignment and distribution specifications. Torres et al. [38] independently formalized a notion similar to FCA, but they only consider alignments that are shifts. Our framework is much more general in that we consider arbitrary alignments. In contrast to our framework, their loop bounds are not tight, and the code is not guard-free. The technique they propose can only help rectify the mismatched simple data alignments; in contrast, our goals are far more general.

As future work, we are extending the framework developed here to singular reference matrices and general non-unimodular alignments. We are integrating the techniques of Computational Alignment with flexible ownership rules and XDP [31] directives. We are also developing a parameterized model of parallel machines so that the effectiveness of candidate linear transformations and Computational Alignments can be evaluated quantitatively. We hope this will lead us to a methodology that will allow us to derive near optimal program transformations for any given architecture. It is also interesting to explore other subclasses of Computational Alignment. We are currently working on a global optimization algorithm based on Computational Alignment with the goal of maximizing uniformity across loops.

References

[1] Michael Wolfe. *Optimizing supercompilers for supercomputers.* The MIT Press, 1990.

[2] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[3] J.R. Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19, pages 233–246, 1984.

[4] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 308–317, Atlanta, GA, June 1988.

[5] E.H. D'Hollander. Partitioning and labeling of index sets in do loops with constant dependence vectors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 139–144, 1989.

[6] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2), 1974.

[7] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126, University of Maryland, 1992.

[8] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations (technical summary). In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 175–187, San Francisco, CA, June 1992.

[9] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on simd machines. In *Proceedings of the Symposium on frontiers of massively parallel computation*, pages 416–423, 1990.

[10] M.F.P O'Boyle and G.A. Hedayat. Data alignment: Transformations to reduce communication on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference, IEE Press*, Williamsburg, 1992.

[11] J. Li and M. Chen. The data alignment phase in compiling programs for distributed memory machines. *Journal of parallel and distributed computing*, 13:213–221, 1991.

[12] Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, Munich, West Germany, January 1987.

[13] Utpal Banerjee. A theory of loop permutations. In *Proceedings of Second Workshop on Programming Languag es and Compilers for Parallel Computing*, August 1989.

[14] J. Ramanujam. Tiling of iteration spaces for multicomputers. In *Proceedings of the Supercomputing 1992*, pages 179–186, 1992.

[15] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of block algorithms. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991.

[16] F. Irigoin and R. Triolet. Supernode partitioning. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, CA, 1988.

[17] Utpal Banerjee. Unimodular transformations of double loops. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[18] M.E. Wolf and M.S. Lam. An algorithmic approach to compound loop transformation. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[19] D. Kulkarni, K.G. Kumar, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[20] K.G. Kumar, D. Kulkarni, and A. Basu. Generalized unimodular loop transformations for distributed memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, Chicago, MI, July 1991.

[21] K.G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hierarchical parallel machines in polynomial time. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, July 1992.

[22] K.G. Kumar, D. Kulkarni, and A. Basu. Mapping nested loops on hierarchical parallel machines using unimodular transformations. *Journal of Parallel and Distributed Computing*, page (submitted).

[23] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, volume 28, June 1993.

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

[25] D. Kulkarni and M. Stumm. Global optimizations by computational alignment. In *In preparation.*

[26] A. Schrijver. *Theory of linear and integer programming.* Wiley, 1986.

[27] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 26, pages 39–50, Williamsburg, VA, April 1991.

[28] J. Ramanujam. Non-singular transformations of nested loops. In *Supercomputing 92*, pages 214–223, 1992.

[29] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the fortran d programming system. Technical Report CRPC-TR91121, Dept of computer Science, Rice University, 1991.

[30] HPF Forum. Hpf: High performance fortran language specification. Technical report, HPF Forum, 1993.

[31] V. Bala, J. Ferrante, and L. Carter. Explicit data placement (xdp): A methodology for explicit compile-time representation and optimization of data movement. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 28, pages 139–149, San Diego, CA, July 1993.

[32] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proceedings of the Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[33] F.P. Preparata and M.I. Shamos. *Computational Geometry an Introduction.* Springer-verlag, 1985.

[34] V. Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 41–53, Portland, OR, June 1989.

[35] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, 1993.

[36] P. Feautrier. Array expansion. In *Proceedings of the 1988 International Conference on Supercomputing*, 1988.

[37] W. Li and K. Pingali. Access normalization: loop restructuring for numa compilers. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[38] J. Torres, E. Ayguade, J. Labarta, and M. Valero. Align and distribute-based linear loop transformations. In *Proceedings of Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, 1993.