# Synthesizing Transformations for Locality Enhancement of Imperfectly-nested Loop Nests

Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali
Department of Computer Science
Cornell University
Ithaca, NY 14853

{nawaaz,mateev,pingali}@cs.cornell.edu

## ABSTRACT

We present an approach for synthesizing transformations to enhance locality in imperfectly-nested loops. The key idea is to embed the iteration space of every statement in a loop nest into a special iteration space called the *product space*. The product space can be viewed as a perfectly-nested loop nest, so embedding generalizes techniques like code sinking and loop fusion that are used in ad hoc ways in current compilers to produce perfectly-nested loops from imperfectly-nested ones. In contrast to these ad hoc techniques however, our embeddings are chosen carefully to enhance locality. The product space is then transformed further to enhance locality, after which fully permutable loops are tiled, and code is generated. We evaluate the effectiveness of this approach for dense numerical linear algebra benchmarks, relaxation codes, and the tomcatv code from the SPEC benchmarks.

## 1. BACKGROUND AND PREVIOUS WORK

Sophisticated algorithms based on polyhedral algebra have been developed for determining good sequences of linear loop transformations (permutation, skewing, reversal and scaling) for enhancing locality in perfectly-nested loops[1]. Highlights of this technology are the following. The iterations of the loop nest are modeled as points in an integer lattice, and linear loop transformations are modeled as nonsingular matrices mapping one lattice to another. A sequence of loop transformations is modeled by the product of matrices representing the individual transformations; since the set of nonsingular matrices is closed under matrix product, this means that a sequence of linear loop transformations can be represented by a nonsingular matrix. The problem of finding an optimal sequence of linear loop transformations is thus reduced to the problem of finding an integer matrix that satisfies some desired property, permitting the full machinery of matrix methods and lattice theory to be applied to the problem of locality-enhancement [2; 4; 23; 17; 21; 3].

[0]This work was supported by NSF grants CCR-9720211, EIA-9726388, ACI-9870687,EIA-9972853.

[1]A perfectly-nested loop is a set of loops in which all assignment statements are contained in the innermost loop.

```
for t = 1,T
  for i1 = 2,N-1
    for j1 = 2,N-1
S1:   L(i1,j1) = (A(i1,j1+1) + A(i1,j1-1)
              + A(i1+1,j1) + A(i1-1,j1)) / 4
    end
  end
  for i2 = 2,N-1
    for j2 = 2,N-1
S2:   A(i2,j2) = L(i2,j2)
    end
  end
end
```

Figure 1: Jacobi

This technology is fairly mature, and it has been incorporated into production compilers, enabling these compilers to produce good code for perfectly-nested loop nests. In most programs however, most loop nests are *imperfectly-nested* because one or more assignment statements are contained in some but not all of the loops of the loop nest. For example, important matrix factorizations like Cholesky, LU and QR factorizations [9] are all imperfectly-nested loop nests. An entire procedure, which usually is a sequence of perfectly- or imperfectly-nested loop nests, can itself be considered to be imperfectly-nested loop nest.

As an example, consider the Jacobi code fragment in Figure 1 which is typical of programs that solve partial differential equations (pde's) by explicit methods. These are called relaxation codes in the compiler literature. They contain an outer loop that counts time-steps; in each time-step, a smoothing operation (stencil computation) is performed on arrays that represent approximations to the solution to the pde. In the Jacobi code, statements `S1` and `S2` touch the same data in each iteration of the `t` loop; instances of these statements in different iterations of the outer loop touch the same data as well. If the arrays do not fit into cache, this reuse will not be exploited. It is possible to exploit the reuse between statements in a given iteration of the `t` loop if the code is rewritten as in Figure 2. This code can be obtained by peeling the first iterations of the `i1`,`j1` loops and the last iterations of the `i2` and `j2` loops, and then fusing the remaining `i1` and `i2` loops as well as `j1` and `j2` loops. Furthermore, reuse between different iterations of the `t`-loop can be exploited by skewing the resulting `i` and `j` loops by `2*t` and then tiling all three loops; if the arrays are stored in column major order, interchanging the `i` and `j` loops allows the

```
for t = 1,T
  for j1 = 2, N-1
    L(2, j1) = (A(2,j1+1) + A(2,j1-1)
             + A(4,j1) + A(1,j1)) / 4
  end
  for i = 3,N-1
    L(i, 2) = (A(i,3) + A(i,1)
             + A(i+1,2) + A(i-1,2)) / 4
    for j = 3,N-1
      L(i,j) = (A(i,j+1) + A(i,j-1)
               + A(i+1,j1) + A(i-1,j1)) / 4
      A(i-1,j-1) = L(i-1,j-1)
    end
    A(i-1,N-1) = L(i-1,N-1)
  end
  for j2 = 2,N-1
     A(N-1,j2) = L(N-1,j2)
  end
end
```

Figure 2: Fused Jacobi

exploitation of spatial locality as well. These variants are not shown here due to their complexity.

A number of approaches have been proposed for enhancing locality of reference in imperfectly-nested loop nests like Jacobi. The performance improvement that can be obtained by transforming only the perfectly-nested loops in such programs can be quite limited [14]. Special purpose techniques for tiling matrix factorization codes have been proposed by Carr and Kennedy [6]; similar work for relaxation codes has been done by Song and Li [22].

A more general approach used by current commercial compilers is to convert an imperfectly-nested loop nest into a perfectly-nested loop nest if possible by applying transformations like *code sinking*, *loop fusion* and *loop fission* [25; 12], and then using locality enhancement techniques for the resulting maximal perfectly-nested loops. For most programs, there are many ways to do this conversion, and the performance of the resulting code may depend critically on how this conversion is done. For example, certain orders of applying these transformations might lead to code that cannot be tiled, while other orders could result in tilable code [13]. A further complication is that loop fission and fusion are themselves useful in improving data locality of loop nests; for example, loop fusion improves inter-loop-nest reuse as in the Jacobi example.

In this paper, we propose a general approach to locality enhancement of imperfectly-nested loops that builds on the standard approach used for perfectly-nested loops. Our strategy is shown in Figure 3. The iteration space of each statement is embedded in a special space called the *product space* using affine embedding functions $F_i$. These embeddings can be chosen so as to improve reuse in the program, and they generalize transformations like loop fusion and loop fission that have been used in the literature for locality enhancement. Furthermore, the product space itself can be viewed as a perfectly-nested loop nest (although one which has many redundant dimensions as we discuss later in this paper), so locality enhancement techniques such as height reduction [16] can be applied to the resulting loop nest; when possible, this loop nest can also be made fully permutable, enabling it to be tiled. Finally, code is generated after projecting out redundant dimensions, using standard techniques from polyhedral algebra; this code generation process may produce imperfectly-nested loop nests if appropriate.

The rest of this paper is organized as follows. Section 2 describes a framework for locality enhancement of imperfectly-nested loop

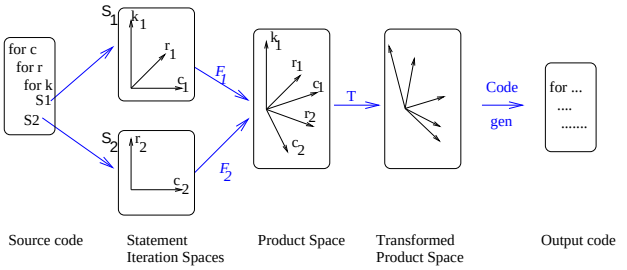



Figure 3: Locality Enhancement of Imperfectly-nested Loop Nests

nests. Section 3 describes the concrete approach to locality enhancement we use in this paper. Because the complete algorithm is complicated, we work through an example in Section 4 to highlight important aspects of our approach. Section 5 gives the details of the algorithm. Section 6 evaluates the effectiveness of this algorithm in enhancing locality of programs on the SGI Octane workstations. We show in that section that our approach automatically performs the transformations on the Jacobi code fragment discussed above. Finally, Section 7 compares our approach with other approaches in the literature.

# 2. AN ABSTRACT MODEL OF LOCALITY ENHANCEMENT

This section gives an abstract view of how program transformations and locality enhancement can be modeled by polyhedral methods.

## 2.1 Program Execution Model

A program is assumed to consist of statements contained in a sequence of perfectly and imperfectly-nested loop nests. All loop bounds and array access functions are assumed to be affine functions of surrounding loop indices. We will use $\mathtt{S}_1$, $\mathtt{S}_2$, ..., $\mathtt{S}_n$ to name the statements in the program in syntactic order. A *dynamic instance* of a statement $\mathtt{S}_k$ refers to a particular execution of the statement for a given value of index variables $\vec{i}_k$ of the loops surrounding it, and is represented by $\mathtt{S}_k(\vec{i}_k)$.

Program execution induces a total order on the dynamic instances of a statement $\mathtt{S}_k$. This *statement execution order* is modeled by the *statement iteration space*, denoted by $\mathcal{S}_k$, as follows:

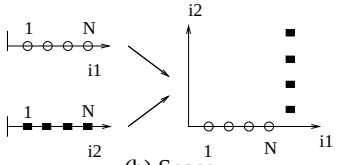
1. Construct a Cartesian space $\mathcal{S}_k$ of dimension equal to the number of loops surrounding $\mathtt{S}_k$.
2. Map dynamic instances of $\mathtt{S}_k$ to $\mathcal{S}_k$ so that the following conditions are satisfied:
   (a) At most one statement instance is mapped to a point in the space $\mathcal{S}_k$.
   (b) If the points in space $\mathcal{S}_k$ are traversed in lexicographic order, and any statement instance mapped to a point is executed when that point is visited, the statement execution order is reproduced.

The program execution order of a code fragment can be modeled in a similar manner by a *program iteration space*, defined as follows:

1. Let $\mathcal{P}$ be a $p$-dimensional Cartesian space for some $p$.
2. Embed all statement iteration spaces $\mathcal{S}_k$ into $\mathcal{P}$ using embedding functions $\tilde{F}_k$ which satisfy the following constraints:

```
for i1 = 1, N
S1:  x[i1] = a[i1]
for i2 = 1, N
S2:  x[i2] += b[i2]
```

(a) Code Fragment




(b) Space

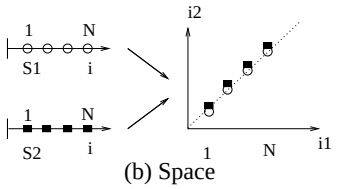
$$F_1(i_1) = \begin{bmatrix} i_1 \\ 0 \end{bmatrix} \quad F_2(i_2) = \begin{bmatrix} N+1 \\ i_2 \end{bmatrix}$$

(c) Embedding Functions

Figure 4: Modeling Program Execution

```
for i = 1, N
S1:  x[i] = a[i]
S2:  x[i] += b[i]
```

(a) Code Fragment




(b) Space

$$F_1(i_1) = \begin{bmatrix} i_1 \\ i_1 \end{bmatrix} \quad F_2(i_2) = \begin{bmatrix} i_2 \\ i_2 \end{bmatrix}$$

(c) Embedding Functions

Figure 5: Reducing Reuse Distances

(a) Each $\tilde{F}_k$ must be one-to-one[2].

(b) If the points in space $\mathcal{P}$ are traversed in lexicographic order, and all statement instances mapped to a point are executed in original program order when that point is visited, the program execution order is reproduced.

The program execution order can therefore be modeled by the pair $(\mathcal{P}, \tilde{\mathcal{F}} = \{\tilde{F}_1, \tilde{F}_2, \ldots, \tilde{F}_n\})$. For the example, the execution order of the code fragment shown in Figure 4(a) can be represented by mapping the statements to a 2-dimensional space as shown in Figure 4(b). The embedding functions for the two statements are shown in Figure 4(c). We will refer to the program execution order as the *original* execution order.

In a similar way, any other execution order of the program can be represented by an appropriate pair $(\mathcal{P}, \mathcal{F})$. We can therefore optimize programs by transforming the original execution order $(\mathcal{P}, \tilde{\mathcal{F}})$ to another execution order $(\mathcal{P}, \mathcal{F})$ that is is somehow "better" for locality and still preserves the semantics of the original program.

## 2.2 Promoting Data Reuse

Consider the data access patterns of the example in Figure 4. Statement instances `S1(i)` and `S2(i)` *exhibit data reuse* because they touch the same memory location `x(i)`. The number of statement instances executed between them is called the *reuse distance*. In the example, the reuse distance is $N-1$. If we represent the execution order for the example by $(\mathcal{P}, \mathcal{F})$, the reuse distance between `S1(i)` and `S2(i)` is proportional to the number of points in $\mathcal{P}$ with statements mapped to them that lie lexicographically between the points to which `S1(i)` and `S2(i)` are mapped. This is because of the initial one-to-one mapping requirement—there are at most a constant number of statement instances (one from each statement) mapped to each point in the space.

Reducing reuse distance increases the likelihood that the data will be in the cache when the second statement instance tries to access it. For our example, we can reduce the reuse distances from $N-1$ to zero by the embedding shown in Figure 5(c). This corresponds to the code obtained by fusing the two loops as shown in Figure 5(a), and can be shown to preserve the semantics of the original code. In general, there are many embeddings that preserve the correctness of a program. We can enhance the locality of a program by picking embeddings that reduces the reuse distances between statement instances while preserving correctness.

[2]Note that instances of *different* statements may get mapped to a single point of the program iteration space.

## 2.3 Reuse Classes

Formally, a reuse exists from instance $\vec{\imath}_s$ of statement $\mathtt{S}_s$ (the source of the reuse) to instance $\vec{\imath}_d$ of statement $\mathtt{S}_d$ (the destination) if the following conditions are satisfied:

1. *Loop bounds*: Both the source and destination statement instances lie within the corresponding iteration space bounds. Since the iteration space bounds are affine expressions of index variables, we can represent these constraints as $B_s * \vec{\imath}_s + b_s \geq 0$ and $B_d * \vec{\imath}_d + b_d \geq 0$ for suitable matrices $B_s, B_d$ and vectors $b_s, b_d$.
2. *Same array location*: Both statement instances reference the same memory location. If we restrict memory references to array references, these references can be written as $A_s * \vec{\imath}_s + a_s$ and $A_d * \vec{\imath}_d + a_d$. Hence the existence of a reuse requires that $A_s * \vec{\imath}_s + a_s = A_d * \vec{\imath}_d + a_d$.
3. *Precedence order*: Instance $\vec{\imath}_s$ of statement $\mathtt{S}_s$ occurs before instance $\vec{\imath}_d$ of statement $\mathtt{S}_d$ in program execution order. If $common_{sd}$ is a function that returns the loop index variables of the loops common to both $\vec{\imath}_s$ and $\vec{\imath}_d$, this condition can be written as $common_{sd}(\vec{\imath}_d) \succeq common_{sd}(\vec{\imath}_s)$ if $\mathtt{S}_d$ follows $\mathtt{S}_s$ syntactically or $common_{sd}(\vec{\imath}_d) \succ common_{sd}(\vec{\imath}_d)$ if it does not, where $\succ$ is the lexicographic ordering relation. This condition can be translated into a disjunction of matrix inequalities of the form $X_s * \vec{\imath}_s - X_d * \vec{\imath}_d + x \geq 0$.

If we express the reuse constraints as a disjunction of conjunctions, each term in the resulting disjunction can be represented as a matrix inequality of the following form.

$$R\begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + r = \begin{bmatrix} B_s & 0 \\ 0 & B_d \\ A_s & -A_d \\ -A_s & A_d \\ X_s & -X_d \end{bmatrix} \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + \begin{bmatrix} b_s \\ b_d \\ a_s - a_d \\ a_d - a_s \\ x \end{bmatrix} \geq 0$$

Each such matrix inequality will be called a *reuse class*, and will be denoted by $\mathcal{R}$ with an appropriate subscript.

The above definition applies to *temporal* reuses where the same array location is accessed by the source and the destination. If the cache line contains more than one array element, then we can also consider *spatial reuse* where the same cache line is accessed by the source and the destination of the reuse. Spatial reuse depends on the storage order of the array.

The conditions for spatial reuse are similar to the ones for temporal reuse, the only difference being that instead of requiring both statement instances to touch the same array location, we require that the

two statement instances touch *nearby array locations that fit in the same cache line*. We can represent this condition as a matrix inequality by requiring the first[3] row of $A_d * \vec{\imath}_d + a_d - A_s * \vec{\imath}_s - a_s$ to lie between 1 and $c-1$, where $c$ is the number of array elements that fit into a single cache line, instead of being equal to 0.

For the example in Figure 4(a), the temporal reuse existing between $\mathtt{S1}(i_1)$ and $\mathtt{S2}(i_2)$ can be represented by the reuse class $\mathcal{R}_1 = \{(i_1, i_2) : 1 \leq i_1, i_2 \leq N, i_1 = i_2\}$. It is straightforward to represent these inequalities as matrix inequalities, but we will not do so to keep the discussion simple. There also exists spatial reuse between these two statements. For a cache line containing 4 array elements it can be represented by the reuse class $\mathcal{R}_2 = \{(i_1, i_2) : 1 \leq i_1, i_2 \leq N, 1 \leq i_2 - i_1 \leq 3\}$.

## 2.4 Legality of Transformations

The original execution order $(\mathcal{P}, \tilde{\mathcal{F}})$ can be legally transformed to another execution order $(\mathcal{P}, \mathcal{F})$ if the latter preserves the semantics of the program. Dependence analysis states that semantics of a program will be preserved if all dependences[4] are preserved under the transformation.

The dependences in a program can be represented by a set of *dependence classes* similar to the reuse classes defined in Section 2.3. Each dependence class captures the constraints between the source ($\vec{\imath}_s$) and destination ($\vec{\imath}_d$) of a dependence that can be represented by a set of linear inequalities $\mathcal{D} : D \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + d \geq 0$.

Let $\mathcal{F} = \{F_1, F_2, \ldots F_n\}$ be a set of embedding functions for the program. We will say that these embedding functions are *legal* if for every $(\vec{\imath}_s, \vec{\imath}_d)$ in a dependence class, the point that $\vec{\imath}_s$ is mapped to in the program iteration space is lexicographically less than the point that $\vec{\imath}_d$ is mapped to. For future reference, we define this formally.

*Definition 1.* Let $\mathcal{F} = \{F_1, F_2, \ldots F_n\}$ be embedding functions that embed the statement iteration spaces of a program into a space $\mathcal{P}$. These embedding functions are said to be *legal* if for every dependence class $\mathcal{D}$ of the program,

$$\forall (\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{D} \;\; F_d(\vec{\imath}_d) \succeq F_s(\vec{\imath}_s)$$

We will refer to the vector $F_d(\vec{\imath}_d) - F_s(\vec{\imath}_s)$ as the *difference vector* for $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{D}$.

## 2.5 Minimizing Reuse Distances

Let $\mathcal{F} = \{F_1, F_2, \ldots F_n\}$ be embedding functions that embed the statement iteration spaces of a program into a space $\mathcal{P}$, and let $\mathcal{R}$ be any reuse class for that program. Consider any reuse pair $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{R}$. Let $Distance(\vec{\imath}_s, \vec{\imath}_d)$ be the number of points in the space $\mathcal{P}$ with statements mapped to them that lie lexicographically between $F_s(\vec{\imath}_s)$ and $F_d(\vec{\imath}_d)$. As we saw in Section 2.2, the reuse distance between $\mathtt{S}_s(\vec{\imath}_s)$ and $\mathtt{S}_d(\vec{\imath}_d)$ is proportional to $Distance(\vec{\imath}_s, \vec{\imath}_d)$. We define $ReuseDistances(\mathcal{P}, \mathcal{F})$ to be the vector of $Distance(\vec{\imath}_s, \vec{\imath}_d)$ for all reuse pairs $(\vec{\imath}_s, \vec{\imath}_d)$ in the program under the execution order $(\mathcal{P}, \mathcal{F})$.

Our goal for locality enhancement is to find legal embeddings $\mathcal{F}_{opt}$ that minimize $\|ReuseDistances(\mathcal{P}, \mathcal{F})\|_X$ for a suitable norm $\|\cdot\|_X$.

[3] For Fortran storage order.

[4] *Dependence* is a temporal reuse in which at least one of the statement instances writes to the common location.

```
 for i = 1, N
  for j = 1, N
S1: c(i,j) = 0
    for k = 1, N
S2:  c(i,j) += a(i,k) * b(k,j)
```

Figure 6: Imperfectly-nested MMM

# 3. A CONCRETE MODEL OF LOCALITY ENHANCEMENT

We now discuss the simplifications we make to develop a practical algorithm. In Section 3.1, we restrict embedding functions to be affine, and define a special space called the *product space* which we argue is the right space for locality enhancement considerations. In Section 3.2, we present our approach to minimizing reuse distances in the product space.

## 3.1 Product Spaces and Embedding Functions

We restrict our embedding functions to be *affine* to permit the use of integer linear programming techniques. Affine embedding functions can be decomposed into their linear and offset parts as follows: $F_k(\vec{\imath}_k) = G_k \vec{\imath}_k + g_k$. We allow symbolic constants in the offset part of the embedding functions.

THEOREM 1. *Let $\mathcal{P}$ be any Cartesian space and let $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ be a set of affine embedding functions $F_k : \mathcal{S}_k \rightarrow \mathcal{P}$. Let $F_k(\vec{\imath}_k) = G_k \vec{\imath}_k + g_k$. The number of independent dimensions of the space $\mathcal{P}$ is equal to the rank of matrix $G = [G_1 G_2 \ldots G_n]$.*

Let $p$ be the sum of the number of dimensions in all statement iteration spaces. Theorem 1 tells us that affine embedding functions cannot utilize more than $p$ dimensions. We therefore use a $p$-dimensional space to model program execution orders.

*Definition 2.* The *product space* for a program is the Cartesian product of all the statement iteration spaces of the statements in that program. The order in which this product is formed is the syntactic order in which the statements appear in the program.

For the imperfectly-nested matrix multiplication code in Figure 6, the iteration space $\mathcal{S}_1$ of statement S1 is a two-dimensional space $i_1 \times j_1$, while the iteration space $\mathcal{S}_2$ of S2 is a three-dimensional space $i_2 \times j_2 \times k_2$. The product space is the five dimensional space $i_1 \times j_1 \times i_2 \times j_2 \times k_2$.

The relationship between statement iteration spaces and the product space is specified by projection and embedding functions. Suppose $\mathcal{P} = \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_n$. Projection functions $\pi_i : \mathcal{P} \rightarrow \mathcal{S}_i$ extract the individual statement iteration space components of a point in the product space, and are obviously linear functions. For the imperfectly-nested matrix multiplication code in Figure 6,

$$\pi_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} I_{2\times 2} & 0 \end{bmatrix},$$

$$\pi_2 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & I_{3\times 3} \end{bmatrix}.$$

*Definition 3.* An *embedding function* $F_i$ maps a point in statement iteration space $\mathcal{S}_i$ to a point in the product space. We consider only those embedding functions $F_i : \mathcal{S}_i \rightarrow \mathcal{P}$ that satisfy the condition $\pi_i(F_i(q)) = q$ for all $q \in \mathcal{S}_i$.

```
for i1 = 1, N
 for j1 = 1, N
  for i2 = 1, N
   for j2 = 1, N
    for k2 = 1, N
S1:  if ((i2==i1)&&(j2==j1)&&(k2==1))
       c(i,j) = 0
S2:  if ((i1==i2)&&(j1==j2))
       c(i,j) += a(i,k) * b(k,j)
```

Figure 7: Original Embeddings for MMM

Intuitively, this condition requires that the statement iteration space of a statement S be mapped to itself in the subspace of the product space corresponding to that statement. This restriction keeps the development simple. Each $F_i$ is therefore one-to-one, but points from two different statement iteration spaces may be mapped to a single point in the product space.

### *3.1.1 Embedding the Original Code*

We show that there always exists a way of embedding the code in the product space that represents the original program execution order.

As an example, consider the code in Figure 6. It is easy to verify that the code in Figure 7 is equivalent, and has the same execution order. Intuitively, the loops in Figure 7 correspond to the dimensions of the product space; the embedding functions for different statements can be read off from the guards in the loop nest:

$$F_1(\begin{bmatrix} i_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} i_1 \\ j_1 \\ i_1 \\ j_1 \\ 1 \end{bmatrix} \quad F_2(\begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix}) = \begin{bmatrix} i_2 \\ j_2 \\ i_2 \\ j_2 \\ k_2 \end{bmatrix}.$$

To preserve the original program execution order, the embedding functions in this example are chosen to satisfy the following conditions: (i) the identity mapping is used for dimensions corresponding to common loops ($i$ and $j$) in the original code; and (ii) mappings for dimensions corresponding to non-common loops are chosen to preserve the original execution order.

In general, we can embed any code into its product space as follows. Let $F_k : \mathcal{S}_k \rightarrow \mathcal{P}$ be the affine function that maps the statement $\mathtt{S}_k$ to the product space. The components of $F_k$ that map into the dimensions of the product space corresponding to statement $\mathtt{S}_j$ are denoted by $F_{k,j}$. Our initial requirement on embedding functions can be summarized by $F_{k,k}(\vec{\imath}_k) = \vec{\imath}_k$.

In Section 2.3 we defined $common_{kl}$ to be a function that returns the loop index variables of the loops common to both $\vec{\imath}_k$ and $\vec{\imath}_l$. Similarly, we define $noncommon_{kl}$ to return the loop index variables of the rest of the loops in $\vec{\imath}_l$. For the example above, we have $common_{12}((i_1, j_1)^T) = (i_1, j_1)^T$, and $noncommon_{12}((i_1, j_1)^T) = k_2$. Let $\min_{\prec}(\vec{\imath})$ and $\max_{\prec}(\vec{\imath})$ return the lexicographically smallest and largest values of the indices of the loops $\vec{\imath}$. For our example, $\min_{\prec}(k_2) = 1$.

For every statement $\mathtt{S}_l$ that occurs syntactically after $\mathtt{S}_k$ in the original program (i.e. $l > k$) we define

$$F_{k,l}(\vec{\imath}_k) = \begin{bmatrix} common_{kl}(\vec{\imath}_k) \\ \min_{\prec}(noncommon_{kl}(\vec{\imath}_k)) \end{bmatrix},$$

and for every statement $\mathtt{S}_l$ that occurs syntactically before $\mathtt{S}_k$ in the original program (i.e. $l < k$) we define

$$F_{k,l}(\vec{\imath}_k) = \begin{bmatrix} common_{kl}(\vec{\imath}_k) \\ \max_{\prec}(noncommon_{kl}(\vec{\imath}_k)) \end{bmatrix}.$$

These embeddings represent the original execution order.

### *3.1.2 Redundant Dimensions in Product Space*

The number of dimensions in the product space can be quite large, and one might wonder if it is possible to embed statement iteration spaces into a smaller space without restricting program transformations. For example, in Figure 7, statements in the body of the transformed code are executed only when `i2 = i2`, so it is possible to eliminate the `i2` loop entirely, replacing all occurrences of `i2` in the body by `i1`. Therefore, dimension $i_2$ of the product space is redundant, as is dimension $j_2$.

There are transformations, however, that use all dimensions of the product space. For example completely fissioned codes, such as the one in Figure 4, need all dimensions of the product space. Since we do not want to restrict transformations unnecessarily, we work with the full product space. Once all embedding functions are determined, redundant dimensions are easy to identify and our algorithm removes them, so there is no performance penalty in the generated code.

## 3.2 Enhancing Locality in the Product Space

Consider a reuse class $\mathcal{R}$ and a reuse pair $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{R}$. The abstract locality enhancement model in Section 2.5 required the minimization of $Distance(\vec{\imath}_s, \vec{\imath}_d)$, which is the number of points in the space $\mathcal{P}$ with statements mapped to them between $F_s(\vec{\imath}_s)$ and $F_d(\vec{\imath}_d)$. Unfortunately, it is not possible to calculate $Distance(\vec{\imath}_s, \vec{\imath}_d)$ efficiently, since there may be points with no statements mapped to them. Instead, we reduce reuse distances as follows.

Consider the *reuse vector* ($\vec{v}$) for the reuse pair $(\vec{\imath}_s, \vec{\imath}_d)$ for a given choice of embedding functions $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$; we will refer to the $j^{th}$ entry of this vector as $v_j$.

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_p \end{bmatrix} = F_d(\vec{\imath}_d) - F_s(\vec{\imath}_s) = \begin{bmatrix} F_{d,1}(\vec{\imath}_d) - F_{s,1}(\vec{\imath}_s) \\ F_{d,2}(\vec{\imath}_d) - F_{s,2}(\vec{\imath}_s) \\ \vdots \\ F_{d,s}(\vec{\imath}_d) - \vec{\imath}_s \\ \vdots \\ \vec{\imath}_d - F_{s,d}(\vec{\imath}_s) \\ \vdots \\ F_{d,n}(\vec{\imath}_d) - F_{s,n}(\vec{\imath}_s) \end{bmatrix}.$$

We say that dimension $j$ *carries reuse* for the reuse pair $(\vec{\imath}_s, \vec{\imath}_d)$ if $v_j \neq 0$. If a dimension carries reuse for some reuse pair in a reuse class $\mathcal{R}$, that dimension is said to carry reuse for that reuse class.

For all reuse pairs $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{R}$, entries corresponding to $F_{d,k}(\vec{\imath}_d) - F_{s,k}(\vec{\imath}_s)$ (for $k \neq s, d$) can be made zero simultaneously (e.g. by choosing $F_{d,k}(\vec{\imath}_d) = F_{s,k}(\vec{\imath}_s) = const$). This may not always be possible for the elements $F_{d,s}(\vec{\imath}_d) - \vec{\imath}_s$ and $\vec{\imath}_d - F_{s,d}(\vec{\imath}_s)$ since the appropriate functions $F_{d,s}$ and $F_{s,d}$ may not exist. We try to make these entries zero; if this does not succeed, we can permute these dimensions of the product space so that they are innermost and tile them. This results in the following strategy:

1. We attempt to make all entries $v_j$ of the reuse vector zero by choosing embedding functions appropriately. Since the dimensions of the embedding functions are independent, we can process each dimension separately. If we succeed in making all entries $v_j = 0$, then the reuse distance is also zero.

2. We reorder the dimensions of the product space so that dimensions for which $v_j = 0$ come first, and dimensions with larger entries come later.

3. We reduce reuse distances further by *tiling* all dimensions $j$ for which the entry $v_j$ of the reuse vector is non-zero.

As an example, consider the code in Figure 4(a) and the reuse from statement `S1(i)` to `S2(i)`. The embeddings in Figure 4(c) result in $v_1 \neq 0, v_2 \neq 0$ and reuse distance of $N-1$, while the embeddings in Figure 5(c) make each entry of the reuse vector zero and result in reuse distance of 0.

# 4. AN EXAMPLE

Before presenting the general locality enhancement algorithm, we illustrate our approach on the program of Figure 6. There are two dependence classes in this example:

1. Dependence class $\mathcal{D}_1 = \{(i_1, j_1, i_2, j_2, k_2) : 1 \leq i_1, j_1, i_2, j_2, k_2 \leq N, i_1 = i_2, j_1 = j_2\}$ is a flow-dependence that arises because statement `S1` writes to a location `c(i,j)` which is then read by statement `S2`.
2. Dependence class $\mathcal{D}_2 = \{(i_2, j_2, k_2, i'_2, j'_2, k'_2) : 1 \leq i_2, j_2, k_2, i'_2, j'_2, k'_2 \leq N, i_2 = i'_2, j_2 = j'_2, k_2 < k'_2\}$ is a flow-dependence that arises because statement `S2` writes to location `c(i,j)` which is then read by this statement in a later $k$ iteration. This dependence also captures the anti- and output-dependences of statement `S2` on itself.

These two classes also represent reuse classes. The program has other reuse classes arising from spatial locality and input dependences, but these are not shown here for simplicity.
As shown in Figure 3, our locality enhancement algorithm will

1. determine affine embedding functions,
2. transform the product space,
3. eliminate redundant dimensions, and
4. decide which dimensions to tile.

The most difficult steps are (1) and (2), and these are interleaved in the algorithm described in Section 5. To simplify the presentation, let us assume for now that an oracle determines the transformation of the product space in Step (2) (we show in Section 5 that interleaving eliminates the need for such an oracle). Therefore, we are left with the problem of determining affine embedding functions. These are determined one dimension at a time by solving a system of linear constraints on the coefficients of the embedding functions for that dimension. These linear constraints describe the requirements that embeddings should (i) result in a legal program, and (ii) minimize reuse distances.
For the running example, we will assume that the oracle tells us that the transformation is the identity transformation, so the product space is left unchanged. Since the product space for this program is the five dimensional space $i_1 \times j_1 \times i_2 \times j_2 \times k_2$, Definition 3 of embedding functions requires that the embedding functions for this program look like the following:

$$F_1\left(\begin{bmatrix} i_1 \\ j_1 \end{bmatrix}\right) = \begin{bmatrix} i_1 \\ j_1 \\ G^3_{i_1} i_1 + G^3_{j_1} j_1 + g^3_N N + g^3_1 \\ G^4_{i_1} i_1 + G^4_{j_1} j_1 + g^4_N N + g^4_1 \\ G^5_{i_1} i_1 + G^5_{j_1} j_1 + g^5_N N + g^5_1 \end{bmatrix}$$

$$F_2\left(\begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix}\right) = \begin{bmatrix} G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1 \\ G^2_{i_2} i_2 + G^2_{j_2} j_2 + G^2_{k_2} k_2 + g^2_N N + g^2_1 \\ i_2 \\ j_2 \\ k_2 \end{bmatrix}$$

The unknowns $G$ and $g$ will be referred to as the *unknown embedding coefficients*.

## 4.1 First Dimension

We first find embedding coefficients for the first dimension $i_1$.

### *Legality*

At the very least, these embeddings must not violate legality. Therefore, as discussed in Section 2.4, the embedding coefficients must satisfy the following constraints.

1. $G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1 - i_1 \geq 0$ for all points in $\mathcal{D}_1$, and
2. $G^1_{i_2} i'_2 + G^1_{j_2} j'_2 + G^1_{k_2} k'_2 + g^1_N N + g^1_1 - (G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1) \geq 0$ for points in $\mathcal{D}_2$.

Standard integer linear programming techniques can be used to convert these constraints into the following system of linear inequalities on the unknown embedding coefficients, as described in the appendix.

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} G^1_{i_2} \\ G^1_{j_2} \\ G^1_{k_2} \\ g^1_N \\ g^1_1 \end{bmatrix} \geq \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (1)$$

### *Minimizing Reuse Distance*

System (1) clearly has many solutions. We need to choose the solution that maximizes reuse. For our running example, consider locality optimization for the reuse that arises because of dependence $\mathcal{D}_1$. To ensure that dimension $i_1$ does not carry reuse for $\mathcal{D}_1$, we require that

$$G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1 - i_1 = 0$$

for all points $(i_1, j_1, i_2, j_2, k_2) \in \mathcal{D}_1$. This condition too can obviously be converted into a system of inequalities on the unknown coefficients of the embeddings. The conjunction of this system and System (1) results in the following solution:

$$G^1_{i_2} = 1, G^1_{j_2} = 0, G^1_{k_2} = 0, g^1_N = 0, g^1_1 = 0$$

Therefore, the first dimensions of the two embedding functions are $F^1_1(i_1, j_1) = i_1$ and $F^1_2(i_2, j_2, k_2) = i_2$. Intuitively, this solution fuses dimensions $i_1$ and $i_2$ of the product space.
Even in our simple example, there are other reuse classes such as $\mathcal{D}_2$. To optimize locality for more than one reuse class, we prioritize the reuse classes heuristically and try to find embedding functions that make entries of the reuse vectors of the highest-priority reuse class equal to zero. Reuse classes are considered in order of priority until all embedding coefficients for that dimension are completely determined. If we assume that reuse class $\mathcal{D}_1$ has highest priority, we see that it completely determines the first dimension of the embedding functions, so no other reuse classes can be considered.

## 4.2 Remaining Dimensions

The remaining dimensions of the embedding functions are determined successively in a manner similar to the first one. The only difference is that some of the dependence classes may already be satisfied by preceding dimensions; these do not have to be considered for legality but only for reducing reuse distances by tiling.
Let us assume that the first $j-1$ dimenions of the embedding functions $\mathcal{F}^{1:j-1}$ have been determined and that we are currently processing the $j^{th}$ dimension of the product space.

### *Legality*

Generalizing the corresponding notion in perfectly-nested loops, we say that a dependence class $\mathcal{D} : D \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + d \geq 0$ is *satisfied* by the first $j-1$ dimensions of the embedding functions $\mathcal{F}^{1:j-1}$ if the difference vector $F_d^{1:j-1}(\vec{\imath}_d) - F_s^{1:j-1}(\vec{\imath}_s)$ is lexicographically positive for all $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{D}$. This means that this dependence will be respected regardless of how the remaining dimensions of the embedding functions are chosen. Therefore it is sufficient to require that for every pair $(\vec{\imath}_s, \vec{\imath}_d)$ in an *unsatisfied* dependence class $\mathcal{D}$,

$$F_d^j(\vec{\imath}_d) - F_s^j(\vec{\imath}_s) = \begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + g_d^j - g_s^j \geq 0 \quad (2)$$

In our running example, it can be shown that none of the dependence classes $\mathcal{D}_1$, $\mathcal{D}_2$ are satisfied by the first dimension of the embedding functions determined above, so both dependence classes must be considered when processing the second dimension.

### *Minimizing Reuse Distance*

Constraining embedding coefficients to minimize reuse distances can be done in an identical manner to the first dimension.

An additional concern in picking coefficients for a dimension other than the first is that we may want to tile that dimension with outer dimensions. Tiling requires that these dimensions be fully permutable. We can ensure this by requiring that the constraint ( 2) holds even for satisfied dependence classes. If the resulting system has solutions, we can pick one that minimizes reuse distances as discussed for the first dimension (note that minimizing reuse distances before we add the tiling constraints might produce embeddings that do not allow tiling). If the resulting system has no solutions, the current dimension cannot be made permutable with outer dimensions, so constraint (2) is dropped for satisfied dependence classes.

Our algorithm produces the following embeddings for the running example:

$$F_1\left(\begin{bmatrix} i_1 \\ j_1 \end{bmatrix}\right) = \begin{bmatrix} i_1 \\ j_1 \\ i_1 \\ j_1 \\ 0 \end{bmatrix} \qquad F_2\left(\begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix}\right) = \begin{bmatrix} i_2 \\ j_2 \\ i_2 \\ j_2 \\ k_2 \end{bmatrix}$$

This embedding allows all five dimensions to be tiled. The code generation algorithm (Section 5.2) determines that $i_2$ and $j_2$ are redundant, and tiles the remaining dimensions.

## 4.3 Putting it All Together

If the transformation on the product space is given, we can obtain embedding coefficients for each dimension successively by constraining them based on (i) legality, (ii) tiling considerations, and (iii) minimizing reuse distances.

The algorithm for formulating legality and tiling constraints for a given dimension $q$ is shown in Figure 8. This algorithm takes the dimension being processed, and the sets of unsatisfied and satisfied dependence classes as input, and returns a linear system $L$ expressing constraints on embedding coefficients.

A small but useful feature of this algorithm is that it supports skewing of a dimension by outer dimensions to enable tiling. Dimension $j$ will be permutable *after skewing* by outer dimensions if the difference vector entries corresponding to the satisfied dependence classes are bounded below by a negative constant. Hence, for every

```
ALGORITHM LegalityConstraints(q, DU, DS ) {
 /*
   q is dimension being processed.
   DU is set of unsatisfied dependence classes.
   DS is set of satisfied dependence classes.
 */

  Construct system Temp constraining the qth dimension
    of every embedding function as follows:

  for each unsatisfied dependence class u ∈ DU
     Add constraints so that each entry in dimension q
       of all difference vectors of u is non-negative;
  for each satisfied dependence class s ∈ DS
     Add constraints so that each entry in dimension q
       of all difference vectors of s + positive α
       is non-negative;

  Use Farkas' lemma to convert system Temp into
    a system L constraining unknown embedding
    coefficients;

  Return L;
 }
```

Figure 8: Formulating Linear System for Legality

```
ALGORITHM PromoteReuse(q,L,RS) {
 /*
   q is dimension being processed.
   L is a system constraining unknown embedding
        coefficients.
   RS is set of prioritized reuse classes.
 */


L'= L
for every reuse class R in RS in priority order
 {
    Z := System constraining unknown embedding function
         coefficients so qth dimension entries of
         all reuse vectors of class R is zero

    if (L' ∩ Z ≠ ∅)
     {
         L' := L' ∩ Z
     }
 }

 return any set of coefficients satisfying L';
}
```

Figure 9: Formulating Linear Systems for Promoting Reuse

pair $(\vec{\imath}_s, \vec{\imath}_d)$ in all satisfied dependence classes $\mathcal{D}$, we require that

$$\begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + g_d^j - g_s^j + \alpha \geq 0, \quad \alpha \geq 0,$$

where $\alpha$ is an additional variable introduced into the system. The solution with the smallest $\alpha$ allows us to make dimension $j$ permutable with the outer dimensions using a minimum amount of skewing. If $\alpha$ can be chosen to be 0, the dimension is permutable with outer dimensions without skewing.

Figure 9 shows how such a linear system is further constrained to determine coefficients for good locality. This algorithm assumes that reuse classes have been sorted in decreasing order of priority using some heuristic. In our implementation, we rank reuse classes by estimating the number of reuse pairs in each class.

```
ALGORITHM LocalityEnhancement

Q  := Set of dimensions of product space;
DU := Set of unsatisfied dependence classes
            (initialized to all dependence classes);
DS := Set of satisfied dependence classes
            (initialized to empty set);
RS := Set of reuse classes of the program
            (sorted by priority);
j  := Current dimension in transformed product space
            (initialized to 1);

while (Q is non-empty)
 {
   for each q in Q
    {
       L = LegalityConstraints(q, DU, DS);
       if system L has solutions
        {   Embedding coefficients for dimension j =
              PromoteReuse(q,L,RS);
            Update DS and DU;
            Delete q from Q;
            j = j + 1;
        }
    }

   // No more dimensions q can be added to current band.
   // Start a new band of fully permutable loops.

   DS := empty set;
 }

Apply Algorithm  DimensionOrdering to the dimensions;
Eliminate redundant dimensions;
Tile permutable dimensions with non-zero ReusePenalty;
```

Figure 10: Algorithm to Enhance Locality

# 5. ALGORITHM

Figure 10 shows the complete locality enhancement algorithm.

## 5.1 High-level Structure

Our algorithm interleaves the determination of the transformation for the product space with the determination of embedding coefficients for each dimension, and finds bands of fully permutable dimensions.

Each iteration of the inner `for each` $q$-loop tries to find a dimension $q$ of the product space which can be permuted into position $j$ of the transformed product space. The legality of this permutation is determined by a call to procedure `LegalityConstraints` in Figure 8 which attempts to find legal embeddings for dimension $j$ which permit this dimension to be permuted with dimensions in the same band. If this procedure succeeds, we call procedure `PromoteReuse` in Figure 9 to choose embeddings with good locality. We drop out of the `for` loop when no more dimensions of the product space can be added to the current fully permutable band. All satisfied dependences are then dropped from further consideration, and a new fully permutable band is started. The algorithm terminates when all dimensions of the product space have been mapped into the transformed space.

### *Reordering of Dimensions*

When constructing bands, the algorithm does not try to optimize the order of dimensions within a band since it adds dimensions to bands in arbitrary order. Since arbitrary order may not be best for locality, we need to reorder dimensions after all embedding coefficients have been determined. This is similar to the problem of choosing a good order for loops in a fully permutable loop nest, and any of the techniques in the literature can be used. Here we present a simple heuristic similar to *memory order* [12]. We reorder dimensions of the product space so that the dimensions with most unsatisfied reuses come last. For each dimension $j$ of the product space, we define the *reuse penalty* of that dimension with respect to embedding functions $\{F_1^j, F_2^j, \ldots, F_n^j\}$ to be the number of reuse pairs in the classes for which the dimension carries reuse.

$$ReusePenalty(j, \mathcal{F}) = \sum_{\mathcal{R} \text{ unsatisfied}} \|\mathcal{R}\|$$

where $\|\mathcal{R}\|$ is the number of reuse pairs in reuse class $\mathcal{R}$. Clearly sorting dimensions in $ReusePenalty$ order is not always legal. Figure 11 shows an algorithm that finds a nearby legal permutation. Intuitively, algorithm `DimensionOrdering` tries to order dimensions greedily so that the dimension with the smallest $ReusePenalty$ is outermost if that is legal. Otherwise, it checks whether the dimension with next smallest $ReusePenalty$ can be placed outermost. Once it finds a dimension to place outermost, it repeats the process with the remaining dimensions. It is easy to see that the algorithm will always produce a legal ordering of the dimensions, and that it will pick the $ReusePenalty$ order if that is legal.

For the running example in Figure 6, our algorithm places all five dimensions of the product space in a single fully permutable band. It then picks the dimension order $j_1 \times j_2 \times k_2 \times i_1 \times i_2$.

```
ALGORITHM DimensionOrdering

RPO  = {i1, i2, ... ip}  // ReusePenalty order
NRPO = ∅                 // nearby permutation

m = p  // number of dimensions left to process
k = 0   // number of dimensions processed
while  RPO ≠ ∅
 {
    for dimension j = 1,m
     {
        l = ij ∈ RPO
        Let NRPO = {i1', i2', ... ik'}
        if {i1', i2', ... ik', l} is legal
         {  NRPO = {i1', i2', ... ik', l}
            RPO  = RPO − {l}
            m = m − 1
            k = k + 1
            continue while loop
         }
     }
 }
```

Figure 11: Determining Dimension Ordering

## 5.2 Code Generation

### *Removing Redundant Dimensions*

Let $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ be the set of embedding functions and let $F_k(\vec{\imath}_k) = G_k\vec{\imath}_k + g_k$. As discussed in Section 3, any dimension $j$ for which the $j^{th}$ row $G^j$ of the matrix $G = [G_1, G_2, \ldots, G_n]$ is linearly dependent on other rows is redundant. Once a good legal ordering of the dimensions is determined, our algorithm identifies all dimensions $j$ for which $G^j$ is linearly dependent on previous dimensions, and eliminates those dimensions.

For our running example, dimensions $j_2$ and $i_2$ are redundant as $G^{j_2} = G^{j_1}$ and $G^{i_2} = G^{i_1}$.

### *Tiling*

All dimensions with non-zero $ReusePenalty$ will benefit from tiling. Each fully permutable band is tiled (after individual dimensions are skewed by outer loops if necessary).

Heuristics have been proposed for choosing tile sizes for perfectly-nested loops [7; 21; 5], and any of them can be used with our frame-

```
//tile counter loops
for t1 = 1,N, B
 for t2 = 0, N, B
  for t3 = 1, N, B
    //iterations within a tile
    for j = t1, min(t1+B-1,N)
     for k = t2, min(t2+B-1,N)
      for i = t3, min(t3+B-1,N)
        if (k==0)
          c(i,j) = 0
        c(i,j) += a(i,k) * b(k,j)
```

Figure 12: Locality-optimized MMM before Code Simplification

work. For the running example, $j_1$, $k_2$, and $i_1$ are not redundant and exhibit reuse, therefore we decide to tile all of them. They are all in the same band and do not require skewing. The resulting tiled code is shown in Figure 12. The *min*'s, *max*'s and conditionals within the loop body are removed by the code generation process using standard techniques from polyhedral algebra.

# 6. EXPERIMENTAL RESULTS

In this section, we present preliminary results from our implementation for three important codes—Cholesky factorization, Jacobi kernel, and the tomcatv SPECfp benchmark. All experiments were run on an SGI Octane workstation based on a R12000 chip running at 300MHz with 32 KB first-level data cache and an unified second-level cache of size 2 MB (both caches are two-way set associative). We present the following performance numbers for each code:

1. Performance of code produced by the SGI MIPSPro compiler (Version 7.2.1) with the "-O3" flag turned on. At this level of optimization, the SGI compiler applies the following set of transformations to the code—it converts imperfectly-nested loop nests to *singly nested loops* (SNLs) by means of fission and fusion and then applies transformations like permutation, tiling and software pipelining inner loops [24].

2. Performance of code produced by an implementation of the techniques described in this paper, and then compiled by the SGI MIPSPro compiler with flags "-O3 -LNO:blocking=off" to disable further tiling by the SGI compiler.

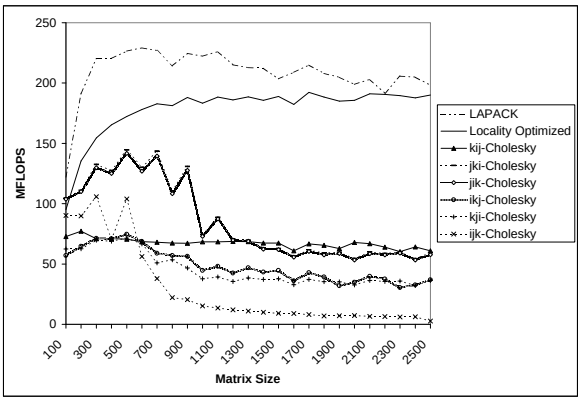
For Cholesky factorization, we also present performance of vendor supplied LAPACK library routine running on top of native BLAS.

Our tile size selection algorithm is still being implemented, so we tiled all codes with a fixed block size of 40. Our experiments show that there are no significant differences in performance for block sizes ranging from 20 to 100. Performance for Cholesky factorization is reported in MFLOPS, counting each multiply-add as 1 Flop. For Jacobi and tomcatv, we did not have hand-coded versions as a comparison; in these cases, we report running time.

The performance numbers presented show the benefits of synthesizing a sequence of locality-optimizing transformations instead of searching for that sequence. Even though the SGI MIPSPro compiler implements all the transformations necessary to optimize our benchmarks, it does not find the right sequence of transformations, so the performance of the resulting code suffers. For Cholesky factorization, the performance of our optimized code approaches the performance of hand-written libraries.

```
for k = 1,N
S1: a(k,k) = sqrt(a(k,k))
    for i = k+1,N
S2:     a(i,k) = a(i,k) / a(k,k)
        for j = k+1,i
S3:         a(i,j) -= a(i,k) * a(j,k)
```

(a) kij-Cholesky Factorization




(b) Performance

Figure 13: Cholesky Factorization and its Performance

## 6.1 Cholesky Factorization

Cholesky factorization is used to solve symmetric positive-definite linear systems. Figure 13(a) shows one version of Cholesky factorization called $kij$-$Cholesky$; there are five other versions of Cholesky factorization corresponding to the permutations of the $i$, $j$, and $k$ loops. Figure 13(b) compares the performance of all six versions compiled by the SGI compiler, the hand-optimized LAPACK library routine, and the code produced by our algorithm starting from *any* of the six versions.

The performance of the compiled code varies widely for the six different versions of Cholesky factorization. The $kij$-$Cholesky$ is SNL and the SGI compiler is able to sink and tile two of the three loops ($k$ and $i$), resulting in good L2 cache behavior and best performance for large matrices (about 65 MFLOPS) among the compiled codes. In contrast, the compiler is not able to optimize the $ijk$-$Cholesky$ at all, resulting in the worst performance of about 5 MFLOPS for large matrices. The LAPACK library code performs consistently best at about 200 MFLOPS.

*Our algorithm produces the same locality optimized code independent on which of the six versions we start with.* That is expected as the abstraction that our algorithm uses—statements, statement iteration spaces, dependencies, and reuses—is the same for all six versions of Cholesky factorization.

For the $kij$ version shown here, the algorithm picks the following embeddings (after reordering and removing redundant dimensions):

$$F_1(\begin{bmatrix} k \end{bmatrix}) = \begin{bmatrix} k \\ k \\ k \end{bmatrix} \quad F_2(\begin{bmatrix} k \\ i \end{bmatrix}) = \begin{bmatrix} k \\ k \\ i \end{bmatrix} \quad F_3(\begin{bmatrix} k \\ i \\ j \end{bmatrix}) = \begin{bmatrix} j \\ k \\ i \end{bmatrix}$$

All three dimensions are tiled without skewing. The same code is obtained starting from any of the six versions of Cholesky factorization, and the line marked "Locality Optimized" in Figure 13(b) shows the performance of that code. The code produced by our approach is roughly 3 to 30 times faster than the code produced by the SGI compiler, and it is within 5% of the hand-written LAPACK library code for large matrices.

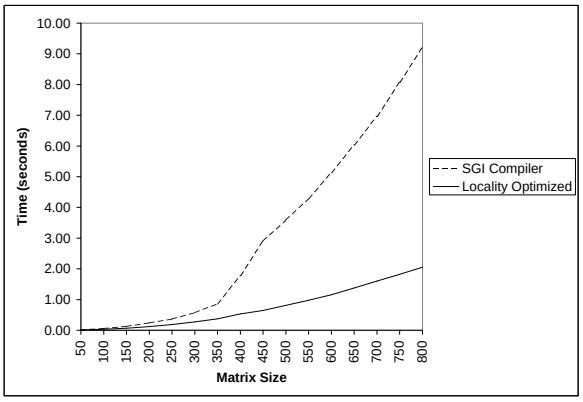



Figure 14: Performance of Jacobi

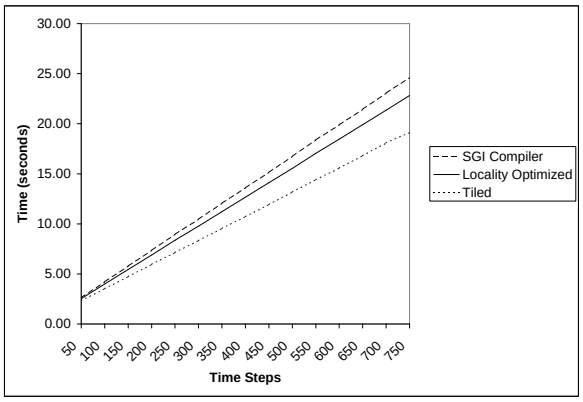



Figure 15: Performance of tomcatv

## 6.2 Jacobi

Our next benchmark is the Jacobi kernel in Figure 1, which was discussed in Section 1. Our algorithm picks embeddings that perform all the optimization steps discussed in Section 1.

$$F_1(\begin{bmatrix} t \\ i \\ j \end{bmatrix}) = \begin{bmatrix} t \\ j \\ i \end{bmatrix} \qquad F_2(\begin{bmatrix} t \\ i \\ j \end{bmatrix}) = \begin{bmatrix} t \\ j+1 \\ i+1 \end{bmatrix}$$

These embeddings correspond to shifting the iterations of the two statements with respect to each other, fusing the resulting `i` and `j` loops respectively, and finally interchanging the `i` and `j` loops. This not only allows us to tile the loops but also benefits the reuses between the two arrays in the two statements, as well as the spatial locality in both statements.

The resulting space cannot be tiled directly, so our implementation chooses to skew the second and the third dimensions by `2*t` before tiling.

Figure 14 shows the execution times for the code produced by our technique and by the SGI compiler for a fixed number of time-steps (100).

## 6.3 Tomcatv

As a final example, we consider the tomcatv code from the SPECfp benchmark suite. The code consists of an outer time loop `ITER` containing a sequence of doubly- and singly-nested loops which walk over both two-dimensional and one-dimensional arrays. The results of applying our technique are shown in Figure 15 for a fixed array size (253 from a reference input), and a varying number of time-steps. Tomcatv is not directly amenable to our technique because it contains an exit test at the end of each time-step. The line marked "Locality Optimized" represents the results of optimizing a single time-step (i.e. the code inside the `ITER` loop) for locality. Treating every basic block as a single statement, our algorithm produces an embedding which corresponds to fusing some of the `J` loops and all the `I` loops. The exploitation of reuse between different basic blocks results in roughly 8% improvement in performance compared to the code produced by the SGI compiler. If we consider the tomcatv kernel without the exit condition[5], our algorithm skews the fused `I` loop by `2*ITER`, and then tiles `ITER` and the skewed `I` loops. The performance of the resulting code (line marked "Tiled") is around 22% better than the original code.

# 7. RELATED WORK AND CONCLUSIONS

The mathematical techniques used in this paper have been used by the systolic array community for scheduling statements in loop nests on systolic arrays [15]. These techniques were extended by Feautrier in his theory of *schedules* in multi-dimensional time [8] which he used for automatic parallelization; related approaches are mappings and affine transforms [10; 18].

For data locality, Kelly and Pugh advocate searching the space of legal transformations using cost models that evaluate the mappings produced [11]. They associate a multi-dimensional mapping with each statement but the range of these mappings is not fixed since there is no analog of our product space. Starting from totally unspecified mappings, they propose to explore the tree of partially specified mappings till they completely specify the mapping, using estimates of the number of cache misses to guide the search. Their estimator targets only reuses with source and destination in the same statement. In particular, reuse between different statements requiring fusion will not be modeled. Though they represent tiling by means of pseudo-linear functions (using `mod` and `div`), they do not include them in their search space of possible transformations and hence do not necessarily find solutions that can be tiled. By using a well-defined space (the product space), we are able to target all reuses by representing reuse distances between statement instances by reuse vectors which we can then minimize dimension by dimension. Furthermore, the product space allows us to impose constraints so that dimensions are permutable—this lets us order the dimensions and tile them to reduce reuse distances further.

A different approach to locality enhancement in imperfectly-nested loops has been taken in *data-centric approaches* such as data shackling [13]. The compiler determines an order in which array elements should be touched, and then schedules code so that all statements that touch a given data element are scheduled to execute when that data item is brought into the cache. Integer linear programming techniques are used to determine if such a schedule is legal. This work has been extended by Pugh and Rosser in their work on iteration space slicing [20]. The data-centric approach can be used to generate code for sparse matrix applications as well [19]. The framework in this paper can be used to generate data-centric code by adding data dimensions to the product space [1].

In conclusion, we have described a systematic approach to locality enhancement of imperfectly-nested loops, and we have shown that it improves performance substantially on codes that are important in practice. Our approach generalizes techniques used in current compilers for locality enhancement of both perfectly-nested and imperfectly-nested loops [4; 23; 17; 25; 24].

[5] The resulting kernel can be tiled speculatively as demonstrated by Song and Li [22].

# 8. REFERENCES


[1] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghil. Compiling imperfectly-nested sparse matrix codes with dependences. Technical Report TR2000-1788, Cornell University, Computer Science, March 2000.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Principle and Practice of Parallel Programming*, pages 39–50, April 1991.

[3] E. Ayguadé and Jordi Torres. Partitioning the statement per iteration space using nonsingular matrices. In *1993 ACM International Conference on Supercomputing*, pages 407–415, Tokyo, July 1993.

[4] Uptal Banerjee. A theory of loop permutations. In *Languages and compilers for parallel computing*, pages 54–74, 1989.

[5] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? In *INTEGRATION, the VLSI Journal*, volume 17, pages 33–51. 1994.

[6] Steve Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing*, 1992.

[7] Jack Dongarra and Robert Schreiber. Automatic blocking of nested loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee, May 1990.

[8] Paul Feautrier. Some efficient solutions to the affine scheduling problem - part ii: multi-dimensional time. *International Journal of Parallel Programming*, December 1992.

[9] Gene Golub and Charles Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[10] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 107–124, Ithaca, New York, August 8–10, 1994. Springer-Verlag.

[11] Wayne Kelly and William Pugh. Selecting affine mappings based on performance estimation. *Parallel Processing Letters*, 4(3):205–209, September 1994.

[12] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *1992 ACM International Conference on Supercomputing*, pages 323–334, Washington, D.C., July 1992. ACM Press.

[13] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Programming Languages, Design and Implementation*. ACM SIGPLAN, June 1997.

[14] Induprakas Kodukula, Keshav Pingali, Robert Cox, and Dror Maydan. Imperfectly nested loop transformations for memory hierarchy management. In *International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[15] S.Y. Kung. *VLSI Array Processors*. Prentice-Hall Inc, 1988.

[16] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 1993.

[17] Wei Li and Keshav Pingali. A singular loop transformation based on non-singular matrices. *International Journal of Parallel Programming*, 22(2), April 1994.

[18] Amy Lim and Monica Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24:445–475, 1998.

[19] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, New Mexico, May 8–11, 2000. To appear.

[20] William Pugh and Evan Rosser. Iteration space slicing for locality. In *Proc. of 12th International Workshop on Languages and Compilers for Parallel Computing, (LCPC99)*, August 1999.

[21] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, October 1992.

[22] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *SIGPLAN99 conference on Programming Languages, Design and Implementation*, June 1999.

[23] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *SIGPLAN 1991 conference on Programming Languages Design and Implementation*, June 1991.

[24] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29*, pages 274–286, Silicon Graphics, Mountain View, CA, 1996.

[25] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.


# APPENDIX

# A. FARKAS' LEMMA

We show how to apply Farkas' Lemma to determine constraints in embeddings.

Consider a dependence class $\mathcal{D} : D \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + d \geq 0$, and a dependence pair $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{D}$. Let $F_d$ and $F_s$ be the embedding functions for the destination and source statements of this dependence, and let the $j^{th}$ dimensions of these functions be $F_d^j = G_d^j \vec{\imath}_d + g_d^j$ and $F_s^j = G_s^j \vec{\imath}_s + g_s^j$.

Suppose that we must choose these unknown coefficients $(G, g)$ so that $F_d^j(\vec{\imath}_d) - F_s^j(\vec{\imath}_s) \geq 0$.

The affine form of Farkas' lemma lets us express the constraints on these coefficients in terms of dependence class coefficients $(D, d)$.

LEMMA 1. *(Farkas' Lemma) Any affine function $f(x)$ which is non-negative everywhere over a polyhedron defined by the inequalities $Ax + b \geq 0$ can be represented as follows:*

$$f(x) = \lambda_0 + \Lambda^T A x + \Lambda^T b, \quad \lambda_0 \geq 0, \Lambda \geq 0,$$

*where $\Lambda$ is a vector of length equal to the number of rows of A. $\lambda_0$ and $\Lambda$ are called the Farkas multipliers.*

Applying Farkas' Lemma to our dependence equations we obtain

$$\left[\begin{array}{cc} -G_s^j & G_d^j \end{array}\right] \left[\begin{array}{c} \vec{\imath}_s \\ \vec{\imath}_d \end{array}\right] + g_d^j - g_s^j = \lambda_0 + \Lambda^T D \left[\begin{array}{c} \vec{\imath}_s \\ \vec{\imath}_d \end{array}\right] + \Lambda^T d,$$
$$\lambda_0 \geq 0, \Lambda \geq 0.$$

Equating coefficients of $\vec{\imath}_s$ and $\vec{\imath}_d$ on both sides, we get

$$\begin{array}{rcl} \left[\begin{array}{cc} -G_s^j & G_d^j \end{array}\right] & = & \Lambda^T D, \\ g_d^j - g_s^j & = & \lambda_0 + \Lambda^T d, \\ \lambda_0 \geq 0, & & \Lambda \geq 0. \end{array}$$

The Farkas multipliers can be eliminated through Fourier-Motzkin projection to give a system of inequalities constraining the unknown embedding coefficients.
As an example, consider the first dimension of the embedding functions for the running example in Section 4. The following conditions must be satisfied:

1. $G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1 - i_1 \geq 0$ for all points $(i_1, j_1, i_2, j_2, k_2)$ in $\mathcal{D}_1 = \{(i_1, j_1, i_2, j_2, k_2) : 1 \leq i_1, j_1, i_2, j_2, k_2 \leq N, i_1 = i_2, j_1 = j_2\}$, and
2. $G^1_{i_2} i'_2 + G^1_{j_2} j'_2 + G^1_{k_2} k'_2 + g^1_N N + g^1_1 - (G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1) \geq 0$ for points $(i_2, j_2, k_2, i'_2, j'_2, k'_2)$ in $\mathcal{D}_2 = \{(i_2, j_2, k_2, i'_2, j'_2, k'_2) : 1 \leq i_2, j_2, k_2, i'_2, j'_2, k'_2 \leq N, i_2 = i'_2, j_2 = j'_2, k_2 < k'_2\}$.

Let us apply Farkas' lemma to the first condition. We have

$$\begin{array}{l} G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1 - i_1 = \\ \quad \lambda_0 + \lambda_1(i_1 - 1) + \lambda_2(N - i_1) + \lambda_3(j_1 - 1) \\ \quad + \lambda_4(N - j_1) + \lambda_5(i_2 - 1) + \lambda_6(N - i_2) + \lambda_7(j_2 - 1) \\ \quad + \lambda_8(N - j_2) + \lambda_9(k_2 - 1) + \lambda_{10}(N - k_2) \\ \quad + \lambda_{11}(i_1 - i_2) + \lambda_{12}(i_2 - i_1) + \lambda_{13}(j_1 - j_2) \\ \quad + \lambda_{14}(j_2 - j_1), \\ \lambda_0, \lambda_1, \ldots, \lambda_{14} \geq 0 \end{array}$$

After equating coefficients on both sides, we get:

$$\begin{array}{rrcl} i_1: & -1 & = & \lambda_1 - \lambda_2 + \lambda_{11} - \lambda_{12} \\ j_1: & 0 & = & \lambda_3 - \lambda_4 + \lambda_{13} - \lambda_{14} \\ i_2: & G^1_{i_2} & = & \lambda_5 - \lambda_6 + \lambda_{12} - \lambda_{11} \\ j_2: & G^1_{j_2} & = & \lambda_7 - \lambda_8 + \lambda_{14} - \lambda_{13} \\ k_2: & G^1_{k_2} & = & \lambda_9 - \lambda_{10} \\ N: & g^1_N & = & \lambda_2 + \lambda_4 + \lambda_6 + \lambda_8 + \lambda_{10} \\ 1: & g^1_1 & = & \lambda_0 - \lambda_1 - \lambda_3 - \lambda_5 - \lambda_7 - \lambda_9 \end{array}$$

Eliminating Farkas multipliers through Fourier-Motzkin projection, we obtain the following constraints on the unknown embedding coefficients:

$$\left[\begin{array}{ccccc} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right] \left[\begin{array}{c} G^1_{i_2} \\ G^1_{j_2} \\ G^1_{k_2} \\ g^1_N \\ g^1_1 \end{array}\right] \geq \left[\begin{array}{c} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{array}\right] \quad (3)$$

Going through the same steps for the second condition, we get

$$\left[\begin{array}{ccccc} 0 & 0 & 1 & 0 & 0 \end{array}\right] \left[\begin{array}{c} G^1_{i_2} \\ G^1_{j_2} \\ G^1_{k_2} \\ g^1_N \\ g^1_1 \end{array}\right] \geq \left[\begin{array}{c} 0 \end{array}\right] \quad (4)$$

Combining conditions (3) and (4), we obtain the system of inequalities shown in Section 4:

$$\left[\begin{array}{ccccc} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right] \left[\begin{array}{c} G^1_{i_2} \\ G^1_{j_2} \\ G^1_{k_2} \\ g^1_N \\ g^1_1 \end{array}\right] \geq \left[\begin{array}{c} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{array}\right]$$