Tiling Imperfectly-nested Loop Nests

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

Abstract

Tiling is one of the more important transformations for enhancing locality of reference in programs. Intuitively, tiling a set of loops achieves the effect of interleaving iterations of these loops. Tiling has been applied only to *perfectly-nested loop nests* which are loop nests in which all assignment statements are contained in the innermost loop. In practice, most loop nests are imperfectly-nested, so existing techniques have limited utility. In this paper, we propose an approach to tiling imperfectly-nested loop nests. The key idea is to embed the iteration space of every statement in the imperfectly-nested loop nest into a special space called the *product space* which is tiled to produce the final code. We evaluate the effectiveness of this approach for dense numerical linear algebra benchmarks, relaxation codes, and the tomcatv code from the SPEC benchmarks. No other approach in the literature can tile all these codes automatically.

1 Background and Previous Work

The memory systems of computers are organized as a hierarchy in which the latency of memory accesses increases by roughly an order of magnitude from one level of the hierarchy to the next. Therefore, a program runs well only if it exhibits enough locality of reference for most of its data accesses to be satisfied by the faster levels of the memory hierarchy. Unfortunately, programs produced by straight-forward coding of most algorithms do not exhibit sufficient locality of reference. The numerical linear algebra community has addressed this problem by writing libraries of carefully hand-crafted programs such as the Basic Linear Algebra Subroutines (BLAS) [16] and LAPACK [2] for algorithms of interest to their community. However, these libraries are useful only when linear systems solvers or eigensolvers are needed, so they cannot be used when explicit methods are used to solve partial differential equations (pde's), for example.

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

(a) Matrix Multiplication

```
//tile counter loops
for t1 = 1,⌈(N/25)⌉
 for t2 = 1,⌈(N/25)⌉
  for t3 = 1,⌈(N/25)⌉
   //iterations within a tile
   for i = (t1-1)*25 +1,min(t1*25,N)
    for j = (t2-1)*25 +1,min(t2*25,N)
     for k = (t3-1)*25 +1,min(t3*25,N)
      c[i,j] = c[i,j] + a[i,k] * b[k,j]
```

(b) Tiled Matrix Multiplication

Figure 1: Tiling Matrix Multiplication

The restructuring compiler community has explored a more *general-purpose* approach in which program locality is enhanced through restructuring by a compiler which does not have any knowledge of the algorithms being implemented by these programs. In principle, such technology can be brought to bear on any program without restriction to problem domain. In practice, most of the work in this area has focused on *perfectly-nested loop nests* that manipulate arrays. A perfectly-nested loop nest is a set of loops in which all assignment statements are contained in the innermost loop; matrix multiplication shown in Figure 1(a) is an example of such a loop nest.

Highlights of the restructuring technology for perfectly-nested loop nests are the following. A loop is said to *carry algorithmic reuse* if the same memory location is accessed by two or more iterations of that

[0] This work was supported by NSF grants CCR-9720211, EIA-9726388 and ACI-9870687.
Corresponding author: nawaaz@cs.cornell.edu

loop for fixed outer loop iterations. In matrix multiplication, the `j` loop carries algorithmic reuse since iterations $(\mathtt{i},\mathtt{j}_1,\mathtt{k})$ and $(\mathtt{i},\mathtt{j}_2,\mathtt{k})$ touch the same location of array `a`. Because of capacity and conflict misses, algorithmic reuse does not necessarily manifest itself as locality of reference during program execution. The likelihood of this manifestation can be increased by permuting a reuse-carrying loop into the innermost position in the loop nest since this transformation decreases the number of memory references that intervene between successive accesses to the memory location of interest. In many programs, there are a number of loops that carry algorithmic reuse; in the matrix multiplication code for example, all three loops carry algorithmic reuse. Since all three loops cannot be moved into the innermost position, the solution is to *tile* these loops, as shown in Figure 1(b). Tiling interleaves iterations of the tiled loops, thereby enabling exploitation of algorithmic reuse in all the tiled loops rather than in just the innermost one [21]. A critical parameter in tiling is the *tile size*, which is the number of iterations of each loop that are performed in each tile. A tile size of 25 was used for all loops in the tiled code in Figure 1.

Tiling changes the order in which loop iterations are performed, so it is not always legal to tile a loop nest. If tiling is not legal, it may be possible to perform *linear loop transformations* like skewing and reversal to enable tiling [1, 3, 13, 17, 19]. Sophisticated heuristics have been proposed for choosing tile sizes [4, 6, 12, 15]. This technology has been incorporated into production compilers such as the SGI MIPSPro compiler, enabling these compilers to produce good code for perfectly-nested loops.

In real programs though, many loop nests are *imperfectly-nested* (that is, one or more assignment statements are contained in some but not all of the loops of the loop nest). Figure 3 shows a loop nest for solving triangular systems with multiple right-hand sides; note that statement `S2` is not contained within the `k` loop, so the loop nest is imperfectly-nested. Cholesky, LU and QR factorizations [8] also contain imperfectly-nested loop nests. A number of approaches have been proposed for enhancing locality of reference in imperfectly-nested loop nests.

The simplest approach is to transform each maximal perfectly-nested loop nest separately. In the triangular solve code in Figure 3, the `c` and `r` loops together, and the `k` loop by itself form two maximal perfectly-nested loop nests. The perfectly-nested loop nest formed by the `c` and `r` loops can be tiled by the techniques described above, but it can be shown that the resulting code performs poorly compared to the code in the LAPACK library which interleaves iterations from all three loops [2].

A more aggressive approach taken in some production compilers such as the SGI MIPSPro compiler is to (i) 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* [22], and then (ii) use locality enhancement techniques for the resulting maximal perfectly-nested loops. In general, there are many ways to do this conversion, and the performance of the resulting code may depend critically on how this conversion is done [10]. Sophisticated heuristics to guide this process were implemented by Wolf et al [20] in the SGI MIPSPro compiler, but our experiments in this paper show that the performance of the resulting code does not approach that of hand-written code in the LAPACK library.

These difficulties led Kodukula et al [10] to propose a novel approach called *data-shackling*. Instead of tiling loop nests, the compiler *blocks* data arrays and chooses an order in which these blocks are brought into the cache; code is scheduled so that all statements that touch a given block of data are executed when that block is brought into the cache, if that is legal. Data-shackling does a good job of enhancing locality in dense numerical linear algebra codes like matrix multiplication, triangular solves with multiple right-hand sides, and matrix factorizations [11]. However, it is not clear how data-shackling can be used for locality enhancement in relaxation codes like Jacobi or Gauss-Seidel that use explicit methods to solve pde's. As we discuss in this paper, these codes make multiple traversals over data arrays, and these traversals must be overlapped to obtain good locality, but it is not clear how to accomplish this with data-shackling.

Recently, Song and Li [18] have proposed techniques for tiling codes like Jacobi. These techniques are very problem-specific since they are geared to programs with a specific structure consisting of an outermost time-step loop that contains a sequence of perfectly-nested loop nests. Their algorithm identifies one loop from each loop nest, fuses these together and skews them with respect to the time-step loop. This transformation strategy is not applicable to codes such as matrix factorizations. Even for relaxation codes, they can only tile loop nests partially, as we explain in Section 4.

Chatterjee et al are exploring the use of space-filling curves to enhance locality in numerical codes [5]. Their goal is to use this idea to write libraries by hand, and there is no effort to generate these blocked codes automatically from high-level algorithms.

In this paper, we propose an approach for automatic tiling of imperfectly-nested loop nests that generalizes the approach used for perfectly-nested loop nests. Our strategy is shown in Figure 2. Each statement `Si` in an imperfectly-nested loop nest is first assigned a unique

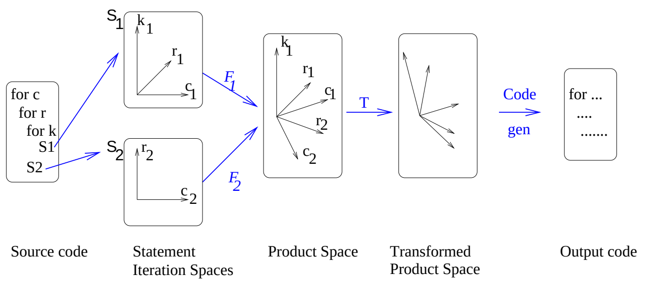


Figure 2: Tiling Imperfectly-nested Loop Nests

iteration space $\mathcal{S}_i$ called the *statement iteration space*. These statement iteration spaces are *embedded* into a large iteration space called the *product space* which is simply the Cartesian product of the individual statement iteration spaces. Embeddings generalize transformations like code-sinking and loop fusion that convert imperfectly-nested loop nests into perfectly-nested ones, and are specified by embedding functions F_i as shown in Figure 2. The product space is further transformed by unimodular transformations to produce a loop nest that can be tiled, if possible. The conditions under which a tilable loop nest can be produced are expressed as matrix inequalities involving the embedding functions F_i and the unimodular transformation T of the product space. In Section 3, we show how embedding functions can be determined for different choices of the unimodular transformation. We are implementing our approach in the SGI MIPSPro compiler, and in Section 4, we present performance results for dense numerical linear algebra codes, relaxation codes and the tomcatv code from the SPEC benchmarks. Finally, we discuss ongoing work in Section 5.

The advantages of our approach are the following. Unlike data-shackling, our approach can be used to enhance locality in codes like Jacobi and tomcatv that must make multiple, overlapped passes over arrays. Unlike the approach of Song and Li, our approach is easily incorporated into a compiler, and it is effective for codes from dense numerical linear algebra. Finally, we show that it outperforms existing techniques in the MIPSPro compiler that also attempt to convert imperfectly-nested loop nests into tilable perfectly-nested loop nests.

2 Product Spaces and Embeddings

The kernel in Figure 3 will be our running example. Triangular systems of equations of the form `Lx = b` where `L` is a lower triangular matrix, `b` is a known vector and `x` is the vector of unknowns arise frequently in applications. Sometimes, it is necessary to solve multiple triangular systems that have the same co-efficient matrix `L`. Such multiple systems can obviously be viewed as computing a matrix `X` that satisfies the equation `LX = B` where `B` is a matrix whose columns are constituted from the right-hand sides of all the triangular systems. The code in Figure 3 solves such multiple triangular systems, overwriting `B` with the solution.

```
for c = 1,M
  for r = 1,N
    for k = 1,r-1
S1:   B(r,c) = B(r,c) - L(r,k)*B(k,c)
S2: B(r,c) = B(r,c)/L(r,r)
```

Figure 3: Triangular Solve with Multiple Right-hand Sides

2.1 Statement Iteration Spaces

We associate a distinct iteration space with each statement in the loop nest, as described in Definition 1.

Definition 1 *Each statement in a loop nest has a* statement iteration space *whose dimension is equal to the number of loops that surround that statement.*

We will use `S1`, `S2`, ..., `Sn` to name the statements in the loop nest in syntactic order. The corresponding statement iteration spaces will be named $\mathcal{S}_1$,$\mathcal{S}_2$,...,$\mathcal{S}_n$. In Figure 3, the iteration space $\mathcal{S}_1$ of statement `S1` is a three-dimensional space $c_1 \times r_1 \times k_1$, while the iteration space $\mathcal{S}_2$ of `S2` is a two-dimensional space $c_2 \times r_2$.

The bounds on statement iteration spaces can be specified by integer linear inequalities. For our running example, these bounds are the following:

$$\begin{array}{rcccc} \mathcal{S}_1 : \mathtt{M} & \geq & c_1 & \geq & 1 \\ \mathtt{N} & \geq & r_1 & \geq & 1 \\ r_1 - 1 & \geq & k_1 & \geq & 1 \end{array} \quad \begin{array}{rcccc} \mathcal{S}_2 : \mathtt{M} & \geq & c_2 & \geq & 1 \\ \mathtt{N} & \geq & r_2 & \geq & 1 \end{array} \qquad (1)$$

An *instance* of a statement is a point within that statement's iteration space.

Since the iteration space bounds are affine expressions of index variables, we can represent the bounds on iteration space $\mathcal{S}_j$ by a matrix expression involving a suitable matrix B_j and vector b_j, as shown below:

$$B_j * i_j + b_j \geq 0$$

2.2 Dependences

We show how the existence of a dependence can be formulated as a set of linear inequalities. As a running example, we use the dependence that arises because statement `S2` writes to location `B(r,c)` which is then read by reference `B(k,c)` in statement `S1`.

A dependence exists from instance i_s of statement `Ss` to instance i_d of statement `Sd` if the following conditions are satisfied.

1. *Loop bounds*: Both source and destination statement instances lie within the corresponding iteration space bounds. This can be expressed as a matrix inequality of the following form:

$$\begin{bmatrix} B_s & 0 \\ 0 & B_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + \begin{bmatrix} b_s \\ b_d \end{bmatrix} \geq 0$$

2. *Same array location*: Both statement instances reference the same array location and at least one of them writes to that location. Since the array references are assumed to be affine expressions of the loop variables, these references can be written as $A_s * i_s + a_s$ and $A_d * i_d + a_d$.
   Hence the existence of a dependence requires that $A_s * i_s + a_s = A_d * i_d + a_d$, which can be expressed as a matrix equation of the following form:

$$\begin{bmatrix} A_s & -A_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + [a_s - a_d] = 0$$

   For our running example, this condition is

$$\begin{aligned} r_2 &= k_1 \\ c_2 &= c_1 \end{aligned}$$

   which can be written as

$$\begin{bmatrix} 0 & 0 & 1 & 0 & -1 \\ 1 & 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ r_1 \\ k_1 \\ c_2 \\ r_2 \end{bmatrix} = 0 \qquad (2)$$

   Systems of equalities can be expressed as inequalities in the obvious way:

$$\begin{bmatrix} A_s & -A_d \\ -A_s & A_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + \begin{bmatrix} a_s - a_d \\ a_d - a_s \end{bmatrix} \geq 0$$

3. *Precedence order*: Instance i_s of statement `Ss` occurs before instance i_d of statement `Sd` in program execution order. If $common_{sd}$ is a function that returns the loop index variables of the loops common to both i_s and i_d, this condition can be written as $common_{sd}(i_d) \succeq common_{sd}(i_s)$ if `Sd` follows `Ss` syntactically or $common_{sd}(i_d) \succ common_{sd}(i_d)$ if it does not, where $\succ$ is the lexicographic ordering relation.
   For the running example, this condition can be written as

$$\begin{bmatrix} c_1 \\ r_1 \end{bmatrix} \succ \begin{bmatrix} c_2 \\ r_2 \end{bmatrix}. \qquad (3)$$

   This condition can be translated into a disjunction of matrix inequalities of the following form:

$$\begin{bmatrix} X_s & -X_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + x \geq 0$$

   It is easy to see that (3) is the disjunction of the following two terms:

$$\begin{bmatrix} -1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} c_1 \\ r_1 \\ k_1 \\ c_2 \\ r_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \geq 0$$

   and

$$\begin{bmatrix} 1 & 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ r_1 \\ k_1 \\ c_2 \\ r_2 \end{bmatrix} + \begin{bmatrix} -1 \end{bmatrix} \geq 0.$$

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

$$D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d = \begin{bmatrix} B_s & 0 \\ 0 & B_d \\ A_s & -A_d \\ -A_s & A_d \\ X_s & -X_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + \begin{bmatrix} b_s \\ b_d \\ a_s - a_d \\ a_d - a_s \\ x \end{bmatrix} \geq 0$$

Each such matrix inequality will be called a *dependence class*, and will be denoted by $\mathcal{D}$ with an appropriate subscript. For our running example in Figure 3, it is easy to show that there are two dependence classes[1]. The first dependence class $\mathcal{D}_1$ arises because statement `S1` writes to a location `B(r,c)` which is then read by statement `S2`; similarly, the second dependence class $\mathcal{D}_2$ arises because statement `S2` writes to location `B(r,c)` which is then read by reference `B(k,c)` in statement `S1`. For simplicity, they are presented as sets of inequalities rather than in matrix notation.

$$\mathcal{D}_1: \quad \begin{array}{rclcl} \mathtt{M} & \geq & c_1 & \geq & 1 \\ \mathtt{N} & \geq & r_1 & \geq & 1 \\ r_1 - 1 & \geq & k_1 & \geq & 1 \\ r_1 & = & r_2 & & \\ c_1 & = & c_2 & & \end{array} \qquad \begin{array}{rclcl} \mathtt{M} & \geq & c_2 & \geq & 1 \\ \mathtt{N} & \geq & r_2 & \geq & 1 \end{array}$$

$$\mathcal{D}_2: \quad \begin{array}{rclcl} \mathtt{M} & \geq & c_1 & \geq & 1 \\ \mathtt{N} & \geq & r_1 & \geq & 1 \\ r_1 - 1 & \geq & k_1 & \geq & 1 \\ k_1 & = & r_2 & & \\ c_1 & = & c_2 & & \end{array} \qquad \begin{array}{rclcl} \mathtt{M} & \geq & c_2 & \geq & 1 \\ \mathtt{N} & \geq & r_2 & \geq & 1 \end{array}$$

[1] There are other dependences, but they are redundant.

2.3 Product Spaces and Embedding Functions

The *product space* for a loop nest is the Cartesian product of the individual statement iteration spaces of the statements within that loop nest. The order in which this product is formed is the syntactic order in which the statements appear in the loop nest.

The relationship between statement iteration spaces and the product space is specified by projection and embedding functions. Suppose $\mathcal{P} = \mathcal{S}_1 \times \mathcal{S}_2 ... \times \mathcal{S}_n$. Projection functions $\pi_i : \mathcal{P} \to \mathcal{S}_i$ extract the individual statement iteration space components of a point in the product space, and are obviously linear functions. For our running example,

$$\pi_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} I_{3\times 3} & 0 \end{bmatrix}$$

$$\pi_2 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & I_{2\times 2} \end{bmatrix}$$

An embedding function F_i on the other hand maps a point in statement iteration space $\mathcal{S}_i$ to a point in the product space. Unlike projection functions, embedding functions can be chosen in many ways. In our framework, we consider only those embedding functions $F_i : \mathcal{S}_i \to \mathcal{P}$ that satisfy the following conditions.

Definition 2 *Let* `Si` *be a statement whose statement iteration space is $\mathcal{S}_i$, and let $\mathcal{P}$ be the product space. An* embedding function $F_i : \mathcal{S}_i \to \mathcal{P}$ *must satisfy the following conditions.*

1. *F_i must be affine.*
2. *$\pi_i(F_i(q)) = q$ for all $q \in \mathcal{S}_i$.*

The first condition is required by our use of integer linear programming techniques. The second condition states that if point $q \in \mathcal{S}_i$ is mapped to a point $p \in \mathcal{P}$, then the component in p corresponding to $\mathcal{S}_i$ is q itself. Therefore, for our running example, we will permit embedding functions like F_1 but not F_2:

$$F_1(\begin{bmatrix} c \\ r \\ k \end{bmatrix}) = \begin{bmatrix} c \\ r \\ k \\ k+1 \\ r+k \end{bmatrix} \quad F_2(\begin{bmatrix} c \\ r \\ k \end{bmatrix}) = \begin{bmatrix} c \\ r+k \\ k \\ k+1 \\ r+k \end{bmatrix}.$$

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. Affine embedding functions can be decomposed into their linear and offset parts as follows: $F_j(i_j) = G_j i_j + g_j$.

We do not require that dependences be satisfied by a lexicographic traversal of the product space. This is because we will allow the product space to be traversed in any unimodular direction, as explained in Section 2.4. For the development in the rest of the paper, it is convenient to define the *embedding matrix* as follows.

Definition 3 *Let* `S1`,`S2`,...,`Sn` *be program statements, and let the linear components of their embedding functions be $G_1, G_2, \ldots, G_n$. The* embedding matrix *is the matrix $G = [G_1 G_2 \ldots G_n]$.*

```
for i = 1, N
    for j = 1, N
S1:     C(i,j) = 0
    for k = 1, N
S2:     B(i,k) = 0
```

(a) Original code

```
for i1 = 1, N
  for j1 = 1, N
    for i2 = 1, N
      for k2 = 1, N
S1:      if ((i2==1)&&(k2==1)) C(i1,j1) = 0
S2:      if ((i1==N)&&(j1==N)) B(i2,k2) = 0
```

(b) Transformed code

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

(c) Embeddings

Figure 4: Embeddings for Loop Fission

2.3.1 Examples of Embeddings

Embeddings can be viewed as a generalization of techniques like code-sinking, loop fission and fusion that are used in current compilers such as the SGI MIPSPro to convert imperfectly-nested loop nests into perfectly-nested ones. Figure 4 illustrates this for loop fission. After loop fission, all instances of statement `S1` in Figure 4(a) are executed before all instances of statement `S2`. It is easy to verify that this effect is achieved by the transformed code of Figure 4(b). Intuitively, the loop nest in this code corresponds to the product space; the embedding functions for different statements can be read off from the guards in this loop nest and are shown in Figure 4(c).

Code sinking is similar and is shown in Figure 5.

2.3.2 Dimension of Product Space

The number of dimensions in the product space can be quite large, and one might wonder if it is possible to

```
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)
```

(a) Original Code

```
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(i1,j1) = 0
S2:        if ((i1==i2)&&(j1==j2))
             C(i2,j2) += A(i2,k2)*B(k2,j2)
```

(b) Transformed code

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

(c) Embeddings

Figure 5: Embeddings for Code Sinking

embed statement iteration spaces into a smaller space without restricting program transformations. For example, in Figure 5(b), statements in the body of the transformed code are executed only when `i2 = i1`, 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. More generally, we have the following result.

Lemma 1 *The rank of the product space is the rank of the embedding matrix G.*

This result is easy to prove. In Figure 5, the embedding matrix is

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

The rank of this matrix is 3, which is also the number of independent dimensions in the product space. In contrast, there are no redundant dimensions in the product space of Figure 4.

In general therefore, it is the embeddings that determine whether there are redundant dimensions in the product space. Since we compute embeddings and transformations simultaneously, we use the full product space to avoid restricting transformations unnecessarily. At the end, our code generation algorithm suppresses redundant dimensions automatically, so there is no performance penalty in the generated code from these extra dimensions.

2.4 Transformed Product Spaces and Valid Embeddings

If p is the dimension of the product space, let $T^{p\times p}$ be a unimodular matrix. Any such matrix defines an order in which the points of the product space are visited. We will say a set of embeddings is *valid* for a given order of traversal of the product space if this traversal respects all dependences. More formally, we have the following definitions.

Definition 4 *The space that results from transforming a product space $\mathcal{P}$ by a unimodular matrix T is called the* transformed product space *under transformation T.*

We want dependences to be satisfied by a lexicographic order of traversal of the transformed product space. To formulate this condition, it is convenient to define the following concept.

Definition 5 *Let $\{F_1, F_2, \ldots F_n\}$ be a set of embedding functions for a program, and let $T^{p\times p}$ be a unimodular matrix. Let*

$$\mathcal{D} : D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d \geq 0$$

be a dependence class for this program. The difference vector *for a pair $(i_s, i_d) \in \mathcal{D}$ is the vector*

$$V_{\mathcal{D}}(i_s, i_d) \equiv T\left[F_d(i_d) - F_s(i_s)\right].$$

The set of difference vectors for all points in a dependence class $\mathcal{D}$ will be called the difference vectors for *$\mathcal{D}$; abusing notation, we will refer to this set as $V_{\mathcal{D}}$.*

The set of all difference vectors for all dependence classes of a program will be called the difference vectors of that program*; we will refer to this set as V.*

With these definitions, it is easy to express the condition under which a lexicographic order of traversal of the transformed product space respects all program dependences.

Definition 6 *Let $T^{p\times p}$ be a unimodular matrix. A set of embedding functions $\{F_1, F_2, \ldots, F_n\}$ is said to be* valid *for T if $v \succeq 0$ for all $v \in V$.*

3 Tiling

We now show how this framework can be used to tile imperfectly-nested loop nests. The intuitive idea is to embed all statement iteration spaces in the product space, and then tile the product space after transforming it if necessary by a unimodular transformation. Tiling is legal if the transformed product space is *fully permutable*—that is, if its dimensions can be permuted arbitrarily without violating dependences. This approach is a generalization of the approach used to tile perfectly-nested loop nests; the embedding step is not required for perfectly-nested loop nests because all statements have the same iteration space to begin with.

3.1 Determining Constraints on Embeddings and Transformations

The condition for full permutability of the transformed product space is the following.

Lemma 2 *Let $\{F_1, F_2, \ldots, F_n\}$ be a set of embeddings, and let T be a unimodular matrix. The transformed product space is fully permutable if $v \geq 0$ for all $v \in V$.*

The proof of this result is trivial: if every entry in every difference vector is non-negative, the space is fully permutable, so it can be tiled. Thus our goal is to find embeddings F_i and a product space transformation T that satisfy the condition of Lemma 2.

Let $\mathcal{D} : D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d \geq 0$ be any dependence class. For affine embedding functions, the condition $v \geq 0$ in Lemma 2 can be written as follows:

$$T \begin{bmatrix} -G_s & G_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + T[g_d - g_s] \geq 0.$$

The affine form of Farkas' Lemma lets us express the unknown matrices T,G_s,g_s,G_d and g_d in terms of D.

Lemma 3 *(Farkas) 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:*

$$\begin{aligned} f(x) &= \lambda_0 + \Lambda^T Ax + \Lambda^T b \\ &\lambda_0 \geq 0, \Lambda \geq 0 \end{aligned}$$

where Λ is a vector of length equal to the number of rows of A. λ_0 and Λ are called the Farkas multipliers.

Applying Farkas' Lemma to our dependence equations we obtain

$$\begin{aligned} T \begin{bmatrix} -G_s & G_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} &+ T[g_d - g_s] \\ &= y + Y^T D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + Y^T d \\ & y \geq 0, Y \geq 0, \end{aligned}$$

where the vector y and the matrix Y are the Farkas multipliers.

Equating coefficients of i_s, i_d on both sides, we get

$$\begin{aligned} T \begin{bmatrix} -G_s & G_d \end{bmatrix} &= Y^T D \\ T[g_d - g_s] &= y + Y^T d \\ y \geq 0, Y &\geq 0. \end{aligned} \tag{4}$$

The Farkas multipliers in System (4) can be eliminated through Fourier-Motzkin projection to give a system of inequalities constraining the unknown embedding coefficients and transformation matrix. Since we require that all difference vector elements be non-negative, we can apply this procedure to each dimension of the product space separately.

Applying the above procedure to all dependence classes results in a system of inequalities constraining the embedding functions and transformation. A fully permutable product space is possible if and only if that system has an integer solution. The set of dimensions for which the equations have a solution will constitute a fully permutable sub-space of the product space.

3.2 Solving for Embeddings and Transformations

In System (4), T is unknown while each G_i is partially specified[2]. To solve such systems, we will heuristically restrict T and solve the resulting linear system for appropriate embeddings if they exist.

3.2.1 Example

Before describing the algorithm, we illustrate this for the running example. The embedding functions for this program can be written as follows:

$$F_1(\begin{bmatrix} c_1 \\ r_1 \\ k_1 \end{bmatrix}) = \begin{bmatrix} c_1 \\ r_1 \\ k_1 \\ f_1^{c_2} \\ f_1^{r_2} \end{bmatrix} \quad F_2(\begin{bmatrix} c_2 \\ r_2 \end{bmatrix}) = \begin{bmatrix} f_2^{c_1} \\ f_2^{r_1} \\ f_2^{k_1} \\ c_2 \\ r_2 \end{bmatrix}$$

where $f_1^{c_2}$ etc. are unknown affine functions that must be determined. Assume that T is the identity matrix. We apply our procedure dimension by dimension to the product space.

Consider the first dimension. We have to ensure two conditions:

1. $f_2^{c_1}(c_2, r_2) - c_1 \geq 0$ for all points in $\mathcal{D}_1$, and
2. $c_1 - f_2^{c_1}(c_2, r_2) \geq 0$ for all points in $\mathcal{D}_2$.

Consider the first condition. Let $f_2^{c_1}(c_2, r_2) = g_{c_2}c_2 + g_{r_2}r_2 + g_M M + g_N N + g_1$. Applying Farkas'

[2] The embedding functions are partially fixed because of condition (2) in Definition 2.

Lemma, we get $f_2^{c_1}(c_2, r_2) - c_1 = \lambda_0 + \lambda_1(M - c_1) + \lambda_2(c_1 - 1) + \cdots + \lambda_{13}(c_1 - c_2) + \lambda_{14}(c_2 - c_1)$ where λ_0, ..., λ_{14} are non-negative[3]. Projecting the λ's out, we find out that the coefficients of $f_2^{c_1}(c_2, r_2)$ must satisfy the following inequalities:

$$\begin{aligned} g_M &\geq 0 \\ g_N &\geq 0 \\ g_{c_2} + g_M &\geq 1 \\ g_{r_2} + g_N &\geq 0 \\ g_{c_2} + 2g_{r_2} + g_M + 2g_N + g_1 &\geq 1 \end{aligned}$$

Similarly, for the second condition, this procedure determines the following constraints:

$$\begin{aligned} g_M &\leq 0 \\ g_N &\leq 0 \\ g_{c_2} + g_M &\leq 1 \\ g_{r_2} + g_N &\leq 0 \\ g_{c_2} + g_{r_2} + g_M + 2g_N + g_1 &\leq 1 \end{aligned}$$

The conjunction of these inequalities gives the solution $f_2^{c_1}(c_2, r_2) = c_2$.

Applying the same procedure to the other dimensions of the product space, we obtain the following set of legal embeddings:

$$\begin{aligned} f_2^{c_1}(c_2, r_2) &= c_2 \\ f_2^{r_1}(c_2, r_2) &\in \{r_2, r_2 + 1\} \\ f_2^{k_1}(c_2, r_2) &\in \{r_2, r_2 - 1\} \\ f_1^{c_2}(c_1, r_1, k_1) &= c_1 \\ f_1^{r_2}(c_1, r_1, k_1) &\in \{r_1, r_1 - 1, k_1, k_1 + 1\}. \end{aligned}$$

In this case, we get more than one solution, and any one of them can be used to obtain a fully permutable product space.

3.2.2 Reversal and Skewing

In general, it may not be possible to find embeddings that make the product space fully permutable (that is, with T restricted to the identity matrix). For such programs, transforming the product space by a non-trivial transformation T may result in a fully permutable space that can be tiled. This is the case for the relaxation codes discussed in Section 4. If our algorithm fails to find embeddings with T restricted to the identity matrix, it tries to find combinations of loop permutation, reversal and skewing for which it can find valid embeddings. The framework presented in Section 3.1 can be used to find these transformations.

[3] There are 14 inequalities that define $\mathcal{D}_1$ in Section 2.2, so there are 14 Farkas multipliers $\lambda_1 \ldots \lambda_{14}$.

Loop reversal for a given dimension of the product space is handled by requiring the entry in that dimension of each difference vector to be non-positive. For a dependence class $\mathcal{D}$, the condition that the j^{th} entry of all of its difference vectors $V_{\mathcal{D}}$ are non-positive can be written as follows:

$$\begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + g_d^j - g_s^j \leq 0$$

which is equivalent to

$$\begin{bmatrix} G_s^j & -G_d^j \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + g_s^j - g_d^j \geq 0.$$

Loop skewing is handled as follows. We replace the non-negativity constraints on the j^{th} entries of all difference vectors in V by linear constraints that guarantee that these entries are bounded below by a negative constant, as follows:

$$\begin{aligned} \begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + g_d^j - g_s^j + \alpha &\geq 0 \\ \alpha &\geq 0 \end{aligned}$$

where α is an additional variable introduced into the system. The smallest value of α that satisfies this system lies at a corner of the polyhedral space defined by the constraints. We can determine α by enumerating the corners of the space. If the system has a solution, the negative entries in the j^{th} entry of all difference vectors are bounded by the value of α. If every difference vector that has a negative value in dimension j, has a strictly positive entry in a dimension preceding j, loop skewing can be used to make all entries in dimension j positive.

3.3 Algorithm

Our algorithm is shown in Figure 6. *The determination of the embedding functions and of the transformation are interleaved, and they are computed incrementally one dimension at a time.* Each iteration of the outer j loop determines one dimension of the transformed product space by determining a dimension q of the embedding functions, and permuting that dimension into the j^{th} position of the transformed product space, reversing that dimension and skewing that dimension by outer dimensions if necessary. If the algorithm fails to find such a dimension q, it attempts to complete the determination of the embedding functions and transformation without regard to the permutability of the remaining dimensions. If this succeeds, the first j-1 dimensions of the transformed product space will be fully permutable and can be tiled.

An important notion in this algorithm is that of a *satisfied* dependence class which is similar to this notion

```
DU := Set of unsatisfied dependence classes
         (initialized to all dependence classes of program)
DS := Set of satisfied dependence classes
         (initialized to empty set)
Q  := Set of dimensions of product space

for dimension j = 1,p of the transformed product space
   for each q in Q
      Construct system 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;
      if system has solutions
         Pick a solution corresponding to smallest α;
         Update DS and DU;
         Delete q from Q and make q the jth dimension
          of the transformed product space;
         Continue j loop;
      endif

      Construct system 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-positive;
         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-positive;
      if system has solutions
         Pick a solution corresponding to smallest α;
         Update DS and DU;
         Delete q from Q and make q the jth dimension
          of the transformed product space;
         Continue j loop;
      endif
   endfor
   //Reach here if no dimension q has a system with a solution.
   //Attempt to satisfy remaining dependence classes
   //without regard to permutability by ignoring already
   //satisfied dependences.
endfor
```

Figure 6: Algorithm to Determine Embeddings and Transformation

in the context of perfectly-nested loop nests [21]. At the j^{th} iteration of the outer loop, we say that a dependence class $\mathcal{D}$ is *satisfied* if all (partially determined) difference vectors $V_{\mathcal{D}}$ are lexicographically positive. Intuitively, a lexicographic traversal of the first j dimensions of the transformed product space is guaranteed to respect all the difference vectors of this dependence class, regardless of how the remaining dimensions of the transformed product space are traversed.

If there are p dimensions in the product space, the body of the q loop is executed $p^2/2$ times in the worst case. Once the transformed product space is determined, we have in effect found a perfectly-nested loop nest with a legal execution order in which some subset of the dimensions are fully permutable and can be tiled. Dependence information for this loop nest can be summarized using directions and distances, and standard techniques for locality enhancement like height reduction [13] can be applied. After this, redundant dimensions are eliminated, fully-permutable loops are tiled, and code is generated using well-understood techniques [1, 9].

3.4 Handling Large Codes

While the algorithm in Figure 6 can be applied to the entire program, it could be quadratic in the size of the product space. Therefore for large codes, we apply the algorithm to more manageable sections created as follows :-

In an initial phase, standard dependence analysis is performed to create the various dependence classes. We create a dependence graph of the program, $G = (V, E)$ where node $v_k \in V$ represents statement S_k and $(v_s, v_d) \in E$ for every dependence class $D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d \geq 0$. We partition the statements into strongly connected components of the dependence graph and process them one component at a time in topological order.For each strongly connected component we create the product space and our algorithm, shown in Figure 6, determines the embedding functions.

The above approach would result in tiling all strongly connected components. Since these are executed in topological order, all dependeces will be preserved.

4 Experimental Results

We are implementing our approach in the SGI MIPSPro compiler. In this section, we present preliminary results from this implementation for five important codes. 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). Wherever possible, we present three sets of performance numbers for a 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 [20].

    1. It converts imperfectly-nested loop nests into an extension of perfectly-nested loop nests called *singly nested loops* (SNLs) by means of fission and fusion.
    2. It applies transforms like permutation and tiling to the SNLs.
    3. It software pipelines the inner loops.

Therefore the lines marked "SGI Compiler" in the graphs correspond to a best-effort code generation by a production compiler applying a large collection of transforms and sophisticated heuristics.

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

5. Performance of hand-coded LAPACK library routine running on top of hand-tuned BLAS written by Mimi Celes at SGI.

The SGI compiler has a cache model which allows it to predict cache misses for a given loop nest, and it uses this model to search for optimal tile sizes. However, the SGI compiler was unable to tile any of the codes described here, so the tile size it picks is irrelevant for this discussion. Our tile size selection algorithm is still being implemented, so we generated code for a range of tile sizes from 10 to 100 along each dimension, and report the best performance in this range.

Performance is reported in MFLOPS, counting each multiply-add as 1 Flop. For some of the codes like tomcatv, we did not have hand-coded versions as a comparison; in these cases, we report running time.

4.1 Triangular Solve

For the running example of triangular solve with multiple right-hand sides, our algorithm determines that the product space can be made fully permutable without reversal or skewing. It chooses the following embeddings:

$$F_1(\begin{bmatrix} c \\ r \\ k \end{bmatrix}) = \begin{bmatrix} c \\ r \\ k \\ c \\ r \end{bmatrix} \quad F_2(\begin{bmatrix} c \\ r \end{bmatrix}) = \begin{bmatrix} c \\ r \\ r \\ c \\ r \end{bmatrix}$$

The fourth and fifth dimensions of the product space are redundant, so they are eliminated and the remaining three dimensions are tiled. All three dimensions are tiled with the same tile size (40). Figure 7 shows performance results for a constant number of right-hand sides (`M` in Figure 3 is 500). The SGI compiler's attempt at code-sinking fails because the `k` loop has a trip count of 0 when the value of the `r` loop index is 1. The performance of code generated by our techniques is upto a factor of 10 better than the code produced by the SGI compiler, but it is still 20% slower than the hand-tuned code in the BLAS library. The high-level structure of the code we generate is similar to that of the code in the BLAS library; further improvements in the compiler-generated code must come from fine-tuning of

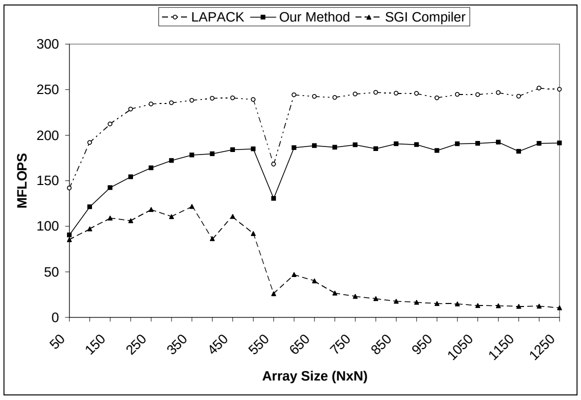


Figure 7: Performance of Triangular Solve

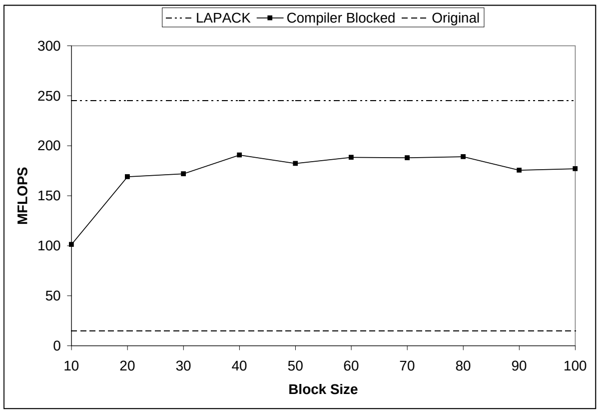


Figure 8: Effect of Bock Size

register tiling and instruction scheduling. As Figure 8 illustrates, the performance is largely independent of the block size as long as the data touched fits in cache.

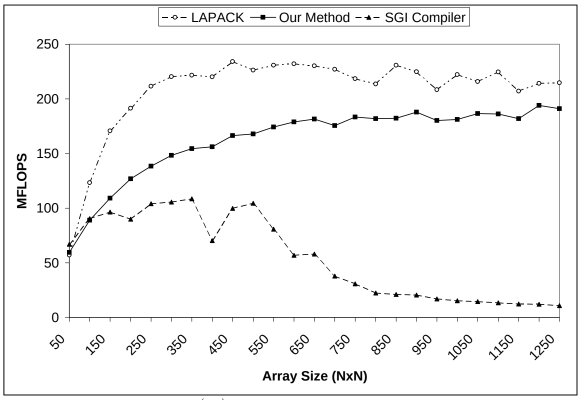
4.2 Cholesky Factorization

Cholesky factorization is used to solve symmetric positive-definite linear systems. Figure 9(a) shows one version of Cholesky factorization called *row-Cholesky* or *ijk-Cholesky*; there are at least five other versions of Cholesky factorization corresponding to the permutations of the three outer loops. Our algorithm correctly determines that the code can be tiled in all the six cases and produces the appropriate embeddings.

For the *ijk* version shown here, the algorithm deduces that all 8 dimensions of the product space can be made fully permutable without reversal or skewing. It picks the following embeddings for the four statements:

```
for i = 1,N
    for j = 1,i-1
        for k = 1,j-1
S1:         a(i,j) = a(i,j) - a(i,k) * a(j,k)
S2:     a(i,j) = a(i,j) / a(j,j)
    for k = 1, i-1
S3:     a(i,i) = a(i.i) - a(i,k) * a(i,k)
s4: a(i,i) = sqrt(a(i,i))
```

(a) Original Code



(b) Performance

Figure 9: Cholesky Factorization and its Performance

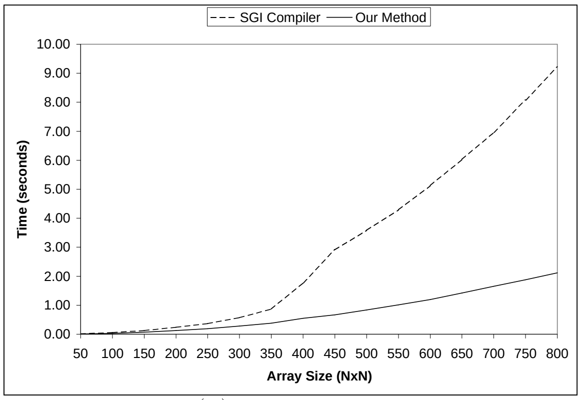
$$F_1(\begin{bmatrix} i \\ j \\ k \end{bmatrix}) = \begin{bmatrix} i \\ j \\ k \\ i \\ k \\ i \\ j \\ i \end{bmatrix} \quad F_2(\begin{bmatrix} i \\ j \end{bmatrix}) = \begin{bmatrix} i \\ j \\ j \\ i \\ j \\ i \\ j \\ i \end{bmatrix}$$

$$F_3(\begin{bmatrix} i \\ k \end{bmatrix}) = \begin{bmatrix} i \\ i \\ i \\ i \\ i \\ i \\ k \\ i \end{bmatrix} \quad F_4(\begin{bmatrix} i \end{bmatrix}) = \begin{bmatrix} i \\ i \\ i \\ i \\ i \\ i \\ i \\ i \end{bmatrix}$$

For these embeddings, the last five dimensions are redundant and are ignored. The remaining three dimensions are tiled. Figure 9(b) shows the result of tiling the three loops with tile-sizes of 60 for varying matrix sizes. The code produced by our approach is roughly 15 times faster than the code produced by the SGI compiler, and it is within 10% of the hand-written LAPACK library code for large matrices.

```
for t = 1,T
  for i = 2,N-1
    for j = 2,N-1
S1:   L(i,j) = (A(i,j+1) + A(i,j-1)
              + A(i+1,j) + A(i-1,j)) / 4
  for i = 2,N-1
    for j = 2,N-1
S2:   A(i,j) = L(i,j)
```

(a) Original Code



(b) Performance

Figure 10: Jacobi and its Performance

4.3 Jacobi

The Jacobi kernel is typical of code in *pde* solvers that use 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*. Most of these applications have imperfectly-nested loop nests. We show the results of applying our technique to the Jacobi kernel shown in Figure 10(a) which uses relaxation to solve Laplace's equation. It requires a non-trivial linear transformation of the product space.

Our algorithm picks an embedding which corresponds intuitively to shifting the iterations of the two statements with respect to each other, and then fusing the resulting `i` and `j` loops.

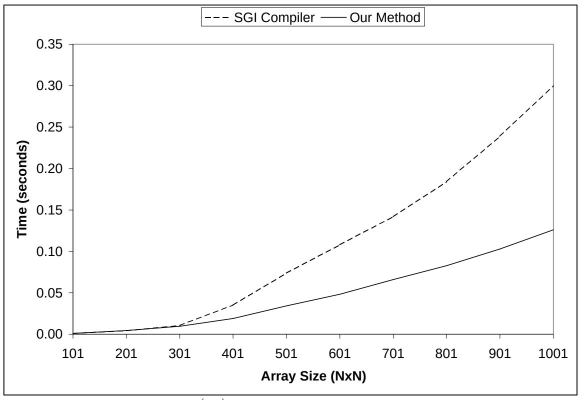
$$F_1(\begin{bmatrix} t \\ i \\ j \end{bmatrix}) = \begin{bmatrix} t \\ i \\ j \\ t \\ i-1 \\ j-1 \end{bmatrix} \quad F_2(\begin{bmatrix} t \\ i \\ j \end{bmatrix}) = \begin{bmatrix} t \\ i+1 \\ j+1 \\ t \\ i \\ j \end{bmatrix}$$

The last three dimensions of the product space are redundant. The resulting product space cannot be tiled directly, so our implementation chooses to skew the second and the third dimensions by `2*t`.

Figure 10(b) shows the execution times for the code

```
for t = 1,T
  for j = 2,N-1
    for i = 2,N-1,2
S1:  U(i,j) = 0.25 * (B(i,j) - U(i-1,j)
              - U(i+1,j) - U(i,j+1) - U(i,j-1))
  for j = 2,N-1
    for i = 3,N-1,2
S2:  U(i,j) = 0.25 * (B(i,j) - U(i-1,j)
              - U(i+1,j) - U(i,j+1) - U(i,j-1))
```

(a) Original Code



(b) Performance

Figure 11: Red-Black Gauss-Siedel and its Performance

produced by our technique and by the SGI compiler for a fixed number of time-steps (100) and a fixed tile-size (30). Our approach improves performance significantly.

Song and Li [18] have an alternative approach that partially tiles the Jacobi kernel. They suggest shifting the second `i` loop with respect to the first, fusing these loops and then skewing the resulting loop by `1*t`. The resulting code does not satisfy dependences, so they propose to eliminate the offending dependences by introducing an additional copy of matrix `A` and performing an even-odd duplication of the code. Finally, they stripmine the `i+t` loop, and interchange the strip counter loop into the outermost position. Compared to our approach, their approach accomplishes only partial tiling since it does not tile the `j` loops. In their paper, they suggest ways in which their approach could be extended to tile the `j` loop as well, but this extension appears to introduce even more complications. Furthermore, the reasoning that a compiler can use to perform this sequence of transformations is unclear.

4.4 Red-Black Gauss-Siedel Relaxation

A more complex relaxation code is the Red-Black Gauss-Seidel code used within multi-grid methods to initialize the values for the next grid level. In Figure 11(a), the value of T is typically small (less than 5). The odd and even rows are processed separately. The arrays are touched twice for each time step. Our implementation finds the following embeddings:

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

These embeddings effectively fuse the odd and even loops together thereby cutting down the memory traffic by a factor of two. The last three dimensions are redundant. The code can be tiled after skewing both the `i` and `j` loops by `1*t`. Figure 11(b) shows the performance of the resulting code when the number of time-steps is set to 5 and a tile size of 20 is chosen.

4.5 Tomcatv

As a final example, we consider the tomcatv kernel from the SPECfp benchmark suite. This kernel is not directly amenable to our technique because it contains an exit test at the end of each time-step, so we removed the exit condition manually. The code, which is too big to be shown here, 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. Treating every basic block as a single statement, our algorithm produces an embedding which corresponds to interchanging the `I` and `J` loops, and then fusing all the `I` loops. The product space is transformed so that the `I` loop is skewed by `2*ITER`, and the `ITER` and skewed `I` loops are tiled. It is not possible to tile the `J` loops in this code because one of the loops walks backwards through some of the arrays. The results of applying the transformation are shown in Figure 12 for a fixed array size (253 from a reference input), fixed tile-size (30) and a varying number of time-steps. The line marked "Our Method" shows a performance improvement of around 18% over the original code. Note that walking the row loop outer-most is bad for spatial locality since FORTRAN arrays are stored in column-major order. If we interchange loops so that the loop walking the entries within the blocked row is inner-most, we get slightly better performance (line marked "Our Method (with interchange)"). Our current implementation does not do this interchange, but it will in the near future. Further improvement is possible by doing a data transformation that transposes all the arrays (line marked "Our Method (with transpose)"). We do not plan to implement data transformations.

As in the Jacobi example, Song and Li skew the `I`

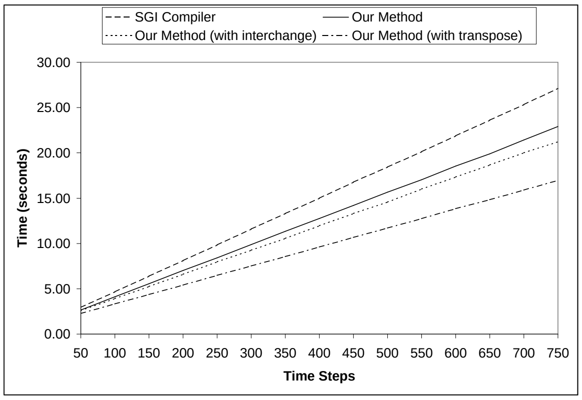


Figure 12: Performance of tomcatv

loops by `1*ITER`, so they need additional storage and odd-even duplication of code to produce legal tiled code.

5 Conclusions

We have presented an approach to tiling imperfectly-nested loop nests, and demonstrated its utility on codes that arise frequently in computational science applications. This approach generalizes techniques used currently to tile perfectly-nested loop nests, and subsumes techniques used in current compilers to convert imperfectly-nested loop nests into perfectly-nested ones for tiling.

Other kinds of embeddings have been used in the literature. For example, Feautrier [7] has solved scheduling problems by embedding statement instances into a one-dimensional space through piecewise affine functions, and searching the space of legal embeddings for one with the shortest length. Kelly et al [9] search a space of affine mappings for programs, using a cost model to choose the best one. The range of these mappings is left undefined to make the framework expressive, but this generality makes it difficult to use. By using the product space, we are able to generalize existing techniques for perfectly-nested loop nests in a straightforward way. Lim et al [14] have used affine partitions to maximize parallelism.

We are implementing our technique in the SGI MIPSPro compiler. In a production setting, compile time is a major concern. The time taken by our implementation on a code like tomcatv appears to be reasonable, so we believe that compile time is not an issue.

Finally, tiling some codes like QR factorization requires exploiting domain-specific information such as the associativity of matrix multiplication. Incorporating this kind of knowledge into a restructuring compiler is critical for achieving the next level of performance from automatic tiling.

References

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

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition.* SIAM, Philadelphia, 1995.

[3] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

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

[5] S. Chaterjee, V. Jain, A. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing (ICS'99)*, Jun 1999.

[6] 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.

[7] Paul Feautrier. Some efficient solutions to the affine scheduling problem - part i: one dimensional time. *International Journal of Parallel Programming*, October 1992.

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

[9] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, February 1995.

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

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

[12] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 8–11, 1991.

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

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

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

[16] Joan McComb Ramesh C. Agarwal, Fred G. Gustavson and Stanley Schmidt. Engineering and Scientific Subroutine Library Release 3 for IBM ES/3090 Vector Multiprocessors. *IBM Systems Journal*, 28(2):345–350, 1989.

[17] Vivek Sarkar. Automatic selection of high order transformations in the IBM ASTI optimizer. Technical Report ADTI-96-004, Application Development Technology Institute, IBM Software Solutions Division, July 1996.

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

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

[20] 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.

[21] M. Wolfe. Iteration space tiling for memory hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

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