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 of *perfectly-nested loop nests* (which are loop nests in which all assignment statements are contained in the innermost loop) is well understood. In practice, many loop nests are imperfectly-nested, so existing compilers use heuristics to try to find a sequence of transformations that convert such loop nests into perfectly-nested ones, but these heuristics do not always succeed. In this paper, we propose a novel 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 single 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 straightforward 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) [22] and LAPACK [3] for algorithms of interest to their community. However, these libraries are useful only when linear systems solvers or eigensolvers are needed; they are not useful when explicit methods are used to solve partial differential equations (pde's), for example.

The restructuring compiler community has explored a more *general-purpose* approach in which program locality is enhanced by a program restructuring compiler which does not have any specific knowledge of the algorithms being implemented by these programs. In principle, this 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[1]. 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 loop for fixed outer loop iterations. Permuting a reuse-carrying loop into the innermost position in the loop nest allows us to exploit the reuse. In many programs, there are a number of loops that carry algorithmic reuse – this can be exploited by *tiling*, which interleaves iterations of the tiled loops, thereby enabling exploitation of algorithmic reuse in all the tiled loops rather than in just the innermost one [28]. Sophisticated heuristics have been proposed for choosing tile sizes [5, 7, 8, 15, 21].

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* such as skewing and reversal to enable tiling [2, 4, 16, 23, 26]. 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 2 shows a loop nest for solving triangular systems with multiple right-hand sides; note that state-

[0] This work was supported by NSF grants CCR-9720211, EIA-9726388, ACI-9870687, EIA-9972853.
0-7803-9802-5/2000/$10.00 (c) 2000 IEEE.

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

ment S2 is not contained within the k loop, so the loop nest is imperfectly-nested. Cholesky, LU and QR factorizations [11] 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 2, 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 BLAS library which interleaves iterations from all three loops [22].

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* [29], 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 whether the resulting code could be tiled depends on how this conversion is done [13]. Sophisticated heuristics to guide this process were implemented by Wolf et al [27] in the SGI MIPSPro compiler, but our experiments show that the performance of the resulting code does not approach that of hand-written code in the LAPACK library [14].

These difficulties led Kodukula et al [13] to propose a technique 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. However, it is not clear how data-shackling can be used for relaxation codes like Jacobi or Gauss-Seidel that make multiple traversals over data arrays. A related approach called *iteration space slicing* was developed by Pugh and Rosser [20], but it does not address tiling.

Special-purpose tiling algorithms focused on particular kinds of imperfectly-nested loops have been proposed in the literature. For example, Carr et al have shown that factorization codes can be tiled after a specific sequence of loop transformations have been performed on them [6]. Recently, Song and Li [24] have proposed a technique for tiling relaxation codes like Jacobi, specific to programs with a particular 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. However, this transformation strategy is not applicable to codes such as matrix factorizations.

In our previous work [1], we developed a general framework to enhance locality in imperfectly-nested loops. Our strategy is shown in Figure 1. Each statement Si in an imperfectly-nested loop nest is first assigned a unique 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 1.

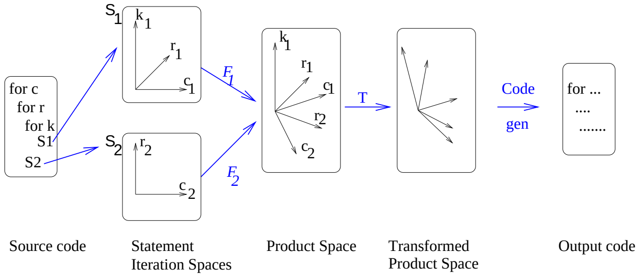


Figure 1: Tiling Imperfectly-nested Loop Nests

In this paper, we show how this framework can be used to tile imperfectly-nested loops. In Section 3, we show how embedding functions that enable tiling can be determined. In Section 3.3, we describe an algorithm that finds nested layers of tilable dimensions of the product-space. Section 3.4 describes how to determine tile sizes for these dimensions. We are currently implementing our approach, and in Section 4, we discuss experimental results obtained by using our approach to tile common benchmarks for the two-level cache hierarchy of an SGI octane. Finally, we discuss ongoing work in Section 5.

The advantages of our approach are the following. By embedding statements in the product space, we abstract away the syntactic structure of the code. Therefore we do not require the to code conform to a particular structure. Secondly, by directly determining embeddings that allow us to perform tiling, we avoid the problem of searching for a sequence of transformations that enables tiling. We know of no other single technique that is capable of tiling all the classes of programs discussed in the paper.

2 Product Spaces and Embeddings

The kernel in Figure 2 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

```
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 2: Triangular Solve with Multiple Right-hand Sides

triangular systems. The code in Figure 2 solves such multiple triangular systems, overwriting B with the solution.

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 2, 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:

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

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

2.2 Dependences

We show how the existence of a dependence can be formulated as a set of linear inequalities.

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. Since the iteration space bounds are affine expressions of index variables, we can represent these constraints as $B_s * i_s + b_s \geq 0$ and $B_d * i_d + b_d \geq 0$ for suitable matrices B_s, B_d and vectors b_s, b_d.
2. *Same array location*: Both statement instances reference the same 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 $A_s * i_s + a_s = A_d * i_d + a_d$.
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.
   This condition can be translated into a disjunction of matrix inequalities of the form $X_s * i_s - X_d * i_d + x \geq 0$.

If we express the dependence constraints as a disjunction of conjunctions, each term in the resulting disjunction can be represented as a matrix inequality 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 2, it is easy to show that there are two dependence classes[2]. 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.

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

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

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.

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

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 $\pi_i : \mathcal{P} \rightarrow \mathcal{S}_i$ extract the individual statement iteration space components of a point in the product space, and are obviously linear functions. For our running example, $\pi_1 = \left[\begin{array}{cc} I_{3\times3} & 0 \end{array} \right]$ and $\pi_2 = \left[\begin{array}{cc} 0 & I_{2\times2} \end{array} \right]$.

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 $F_i : \mathcal{S}_i \rightarrow \mathcal{P}$ 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 *$F_i : \mathcal{S}_i \rightarrow \mathcal{P}$ 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. We will allow symbolic constants in the affine part of the embedding functions. 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. 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: $F_j(i_j) = G_j i_j + g_j$.

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 3 illustrates this for loop fission. After loop fission, all instances of statement `S1` in Figure 3(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 3(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 3(c).

Code sinking is similar and is shown in Figure 4.

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 embed statement iteration spaces into a smaller space without restricting program transformations. For example, in Figure 4(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 can state the following result.

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

(b) Transformed code

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

(c) Embeddings

Figure 3: Embeddings for Loop Fission

Theorem 1 *Let $\mathcal{P}'$ be any space and let $\{F_1, F_2, \ldots, F_n\}$ be a set of affine embedding functions $F_j : \mathcal{S}_j \rightarrow \mathcal{P}'$ satisfying the conditions in Definition 2. Let $F_j(i_j) = G_j i_j + g_j$. The number of independent dimensions of the space $\mathcal{P}'$ is equal to the rank of the matrix $G = [G_1 G_2 \ldots G_n]$.*

In Figure 4, the rank of this matrix

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

is 3, which is also the number of independent dimensions in the product space. The remaining 2 dimensions are redundant.

Corollary 1 *Let $\mathcal{P}$ be the product space.*

1. *Any space $\mathcal{P}'$ bigger than $\mathcal{P}$ has redundant dimensions under any set of affine embedding functions.*

```
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
    if ((i2==i1)&&(j2==j1)&&(k2==1))
S1:   C(i1,j1) = 0
    if ((i1==i2)&&(j1==j2))
S2:   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 4: Embeddings for Code Sinking

2. *There exist affine embedding functions $\{F_1, F_2, \ldots, F_n\}$ for which no dimension of $\mathcal{P}$ is redundant.*

Intuitively, Corollary 1 states that the product space is "big enough" to model any affine transformation of the original code. Furthermore, there are affine transformations that utilize all dimensions of the product space. For example, there are no redundant dimensions in the product space of completely fissioned code, as Figure 3 illustrates.

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 (see Figure 1). More formally, we have the following definitions.

Definition 3 *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.*

For a set of embedding functions $\{F_1, F_2, \ldots F_n\}$ and a transformation matrix T, we model execution of the transformed code by walking the transformed product space lexicographically and executing all statement instances mapped to each point as we visit it. We say that the pair $(\{F_1, F_2, \ldots F_n\}, T)$ defines an *execution order* for the program. For an execution order to be *legal*, a lexicographic order of traversal of the transformed product space must satisfy all dependencies. To formulate this condition, it is convenient to define the following concept.

Definition 4 *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 5 *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 [17, 26]; 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 1 *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 1.

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 1 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 [9].

Lemma 2 *(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

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

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

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

$$\begin{aligned} T \begin{bmatrix} -G_s & G_d \end{bmatrix} &= Y^T D \\ T\,[g_d - g_s] &= y + Y^T d \\ y \geq 0, Y &\geq 0. \end{aligned} \qquad (1)$$

The Farkas multipliers in System (1) 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 a solution. The subset of dimensions for which the equations do have a solution will constitute a fully permutable sub-space of the product space.

3.2 Solving for Embeddings and Transformations

In System (1), T is unknown while each G_i is partially specified[3]. 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:

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

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' 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 $\lambda_0, \ldots, \lambda_{14}$ are non-negative[4]. 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:

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

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

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

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.

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:

$$\begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + g_d^j - g_s^j + \alpha \geq 0, \ \ \alpha \geq 0 \qquad (2)$$

where α is an additional variable introduced into the system. The smallest value of α that satisfies this system can be found by projecting out the other variables and picking the lower bound of α. 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 5. *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 the embedding functions for dimension q of the product space, 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.

The algorithm as shown in Figure 5 does not stop after identifying the outer set of permutable dimensions. Suppose we are trying to determine the j^{th} dimension of the transformed product space. If none of the remaining dimensions of the product space can be made permutable with the outer $j - 1$ dimensions (even allowing reversal and skewing), the algorithm continues and determines a subset of these dimensions that can be made permutable with respect to each other (but not with the outer dimensions). By applying this successively, the algorithm creates a transformed product space that consists of nested layers of permutable dimensions.

An important notion in this algorithm is that of a *satisfied* dependence class which is similar to this notion in the context of perfectly-nested loop nests [28]. 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. In checking whether a particular dimension q of the product space can be made permutable with the outer dimensions, the linear system we construct specifies that all unsatisfied dependence classes DU must have non-negative entries along this dimension[5] while the satisfied dependence

[5] non-positive for the reversal case

```
ALGORITHM DetermineEmbeddings

$Q$  := Set of dimensions of product space
$J$  := Current layer (initialized to 1)
$DU$ := Set of unsatisfied dependence classes
        (initialized to all dependence classes of program)
$DS$ := Set of satisfied dependence classes
        for the current layer (initialized to empty set)
$T$  := Transformation matrix (initialized to Identity)

for dimension $j$ = 1,$p$ of the transformed product space

process_dimension :
   for each $q$ in $Q$

      Construct system $S$ constraining the $qth$ dimension
        of every embedding function as follows:
        for each unsatisfied dependence class $u \in DU$
          Add constraints so that each entry
          in dimension $q$ of all difference vectors
          of $u$ is  non-negative;
        for each satisfied dependence class $s \in DS$
          Add constraints so that each entry
          in dimension $q$ of all difference vectors
          of  $s$ + positive $\alpha$ is non-negative;
      if system has solutions
        Pick a solution corresponding to smallest $\alpha$;
        Update $DS$ and $DU$;
        Delete $q$ from $Q$ and make $q$ the $jth$ dimension
          of the transformed product space;
        Update row $j$ of $T$;
        Continue $j$ loop;
      endif

      // if the previous system does not have a solution
      // check whether reversing the dimension
      // permits solutions
      Construct system $S$ constraining the $qth$ dimension
        of every embedding function as follows:
        for each unsatisfied dependence class $u \in DU$
          Add constraints so that each entry
          in dimension $q$ of all difference vectors
          of $u$ is  non-positive;
        for each satisfied dependence class $s \in DS$
          Add constraints so that each entry
          in dimension $q$ of all difference vectors
          of $s$ - positive $\alpha$ is non-positive;
      if system has solutions
        Pick a solution corresponding to smallest $\alpha$;
        Update $DS$ and $DU$;
        Delete $q$ from $Q$ and make $q$ the $jth$ dimension
        of the transformed product space;
        Update row $j$ of $T$;
        Continue $j$ loop;
      endif
   endfor

   // Reach here if no further dimension of $Q$
   // can be added to the current layer
   $J$  := $J$ + 1 // Start a new layer
   $DS$ := empty set
   goto process_dimension

endfor
```

Figure 5: Determining Embeddings and Transformation

classes DS are allowed to have entries greater than some constant negative[6] α to allow skewing by an outer dimension. This ensures that a solution with skewing is accepted only if it is legal. By choosing a solution that corresponds to minimum α, we choose embedding functions that require the least amount of skewing for successful tiling. If the minimum value of α is 0, then no skewing is required.

When determining the j^{th} dimension of the transformed space, we may fail to find a dimension of the product space to add to our current set of permutable dimensions. If so, we start a new layer of permutable dimensions nested within the outer layers. To do this, we simply need to drop from consideration the satisfied dependence classes (DS).

The correctness of the algorithm follows from this theorem.

Theorem 2 *The algorithm* `DetermineEmbeddings` *has the following properties:*

1. *It always produces embeddings $\{F_1, F_2, \ldots, F_n\}$ and transformation matrix T defining a legal execution order.*
2. *T orders the dimensions of the transformed product space $\mathcal{P}$ into layers J. The dimensions within a layer are fully permutable.*

Note that although in the worst case the algorithm could require $O(p^2)$ executions of the q loop in Figure 5, in practice the number of executions of the q loop is closer to $O(p)$. Each iteration of the q loop needs to perform Fourier-Motzkin elimination which could potentially be exponential; this can be engineered to work well in practice [19].

Once the transformed product space is determined, we have in effect found a perfectly-nested loop nest with a legal execution order. The loops are grouped into layers and loops within each layer are fully permutable within the layer 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 [16] can be applied. After this, redundant dimensions are eliminated, fully-permutable loops are tiled, and code is generated using well-understood techniques [2, 12].

3.3.1 Picking good embedding functions

As far as tiling is concerned, any solution to the system S created by the algorithm in the q loop would allow that dimension of the transformed product space to be fully permutable with the other dimensions in the current layer. If our only concern is to produce tiled code, then any of the solutions will do. But not all solutions are equally effective. For example, consider the code fragment in Figure 6 which will benefit from tiling since the accesses to arrays `A` and `B` in statement `S1` cannot be made unit-stride at the same time.

Two valid embeddings for $T = I$ that allow this code to be tiled are the following:

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

[6] positive α in the reversal case

```
for i1 = 1, N
  for j1 = 1, N
S1: A(i1,j1) = B(j1,i1)

for i2 = 1, N
  for j2 = 1, N
S2: A(i2,j2) = A(i2,j2) + B(i2,j2)
```

Figure 6: To Fuse Or Not To Fuse?

This embedding corresponds to the original program execution order. Tiling the resulting transformed product space would in effect tile the two loop nests separately.

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

   This embedding corresponds to fusing `i1` and `i2` loops and the `j1` and `j2` loops. The resulting fused `i` and `j` loops can now be tiled.

By fusing the two loops, the second solution is able to reduce the distance between the write to array `A` in statement `S1` and the subsequent read in statement `S2`. In fact, in this solution, both the source and destination of this data reuse are mapped to the same point in the product space. The distance between the source and the destination statement instances can be represented by the difference vector for this dependence class. For the first solution, the difference vector corresponding to this dependence is $[N - i_1, N - j_1, i_2 - 1, j_2 - 1]^T$ while it is $[0, 0, 0, 0]^T$ for the second solution. For this code fragment, we prefer the second solution because it exploits the reuse.

If $\{F_1, F_2, \ldots F_n\}$ are the set of embedding functions for a program, and $\mathcal{D} : D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d \geq 0$ is a dependence class for this program, then the difference vector for a pair $(i_s, i_d) \in \mathcal{D}$ is $[F_d(i_d) - F_s(i_s)]$. Clearly, we can reduce the distance between the dependent iterations by reducing each dimension of the difference vector. The reuse is fully exploited if the difference vector is $\vec{0}$. The embedding functions that are able to achieve this satisfy the plane $[F_d(i_d) - F_s(i_s)] = \vec{0}$ for all pairs $(i_s, i_d) \in \mathcal{D}$. If the intersection of this plane with the system of legal solutions (S) is non-empty, then a solution belonging to the intersection will exploit the reuse. We pick a solution that exploits as many reuses as possible, as discussed in detail in a companion paper [1]. For the code fragment shown in Figure 6, this heuristic will pick the second solution.

For our running example, our algorithm picks the following embeddings :-

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

3.4 Determining Tile Sizes

We determine tile sizes by estimating the *data foot-print* (the amount of data touched by a tile), and requiring that it fit into the cache in consideration. We tile the product space separately for each level of the memory hierarchy (we do not tile for a particular level only if the data touched by the tile will not fit into the corresponding cache level).

Our procedure for determining tile sizes has the following steps.

1. For each point in the product-space, find the data accessed by each statement instance mapped to it. Since the mapping from a statement instance to the product-space is one-to-one and affine, the inverse mapping can easily be determined. Since data references are affine, this enables us to calculate the data accessed by each point in the product space.

   In our running example, a point $(x_1, x_2, x_3, x_4, x_5)$ of our transformed product space has the statement instances $S_1(x_1, x_2, x_3)$ (whenever $x_1 = x_4, x_2 = x_5$) and $S_2(x_1, x_2)$ (whenever $x_1 = x_4, x_2 = x_3 = x_5$) mapped to it. Hence the data accessed by this point is $B(x_1, x_2), L(x_2, x_3), B(x_3, x_1)$ from statement S_1 and $B(x_1, x_2), L(x_2, x_2)$ from statement S_2.

2. Group all the data accessed by a product space point into equivalence classes as follows.

   (a) References to different arrays belong in different equivalence classes.

   (b) References to the same array are grouped in the same equivalence class if they can access the same data elements (that is, if the linear parts of the two references are the same).

   For our example, we have four equivalence classes $\{S_1 : B(x_1, x_2), S_2 : B(x_1, x_2)\}$, $\{S_1 : L(x_2, x_3)\}$, $\{S_1 : B(x_3, x_1)\}$ and $\{S_1 : L(x_2, x_2)\}$.

3. From each reference class pick a random reference which will serve as our *representative reference*.

   In our example, our representative references are $B(x_1, x_2)$, $L(x_2, x_3)$,$B(x_3, x_1)$ and $L(x_2, x_2)$.

4. Determine the data touched by each representative reference in a single tile of the transformed product space parameterized by the tile size. We limit ourselves to choosing the same tile size B for all dimensions of the

product space. Determining the number of elements touched by a single reference is straightforward.

In our example, the first three references access B^2 elements in one tile of the transformed product space, while the last reference accesses B elements. The total data foot-print of all the references is thus $3 * B^2 + B$ elements. This must be multiplied by the size in bytes of each element to give the actual memory touched.

5. The data foot-print of all the references must be less than the cache size to avoid capacity misses. This gives us an upper bound on the tile size for each cache level. In order to generate code with fewer `MIN`'s and `MAX`'s, we ensure that the tile size at each level is a multiple of the tile size at the previous level.

The above formulation makes the following simplifications –

1. All tiles of the transformed product space have the same data foot-print. This is a conservative assumption, since it results in adding the footprints of all references from all statements to obtain the data accessed at a single product space point.

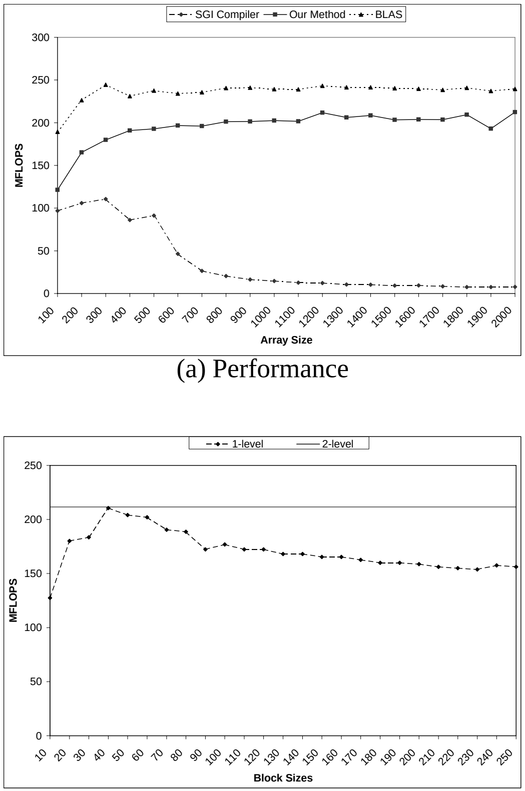
2. Boundary effects are ignored, which is justifiable for large arrays and loop bounds.

3. Conflict misses are ignored. Various techniques have been developed to find tile sizes that avoid some forms of conflict misses [7, 10, 15], but we do not use them in our current implementation.

4 Experimental Results

In this section, we present results from our implementation for four 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 [27] – 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.
2. 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.



(a) Performance

(b) Variation with tile size

Figure 7: Performance of Triangular Solve

3. Performance of hand-coded LAPACK library routine running on top of hand-tuned BLAS.

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:

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

The fourth and fifth dimensions of the product space are redundant, so they are eliminated and the remaining three dimensions are tiled. Our algorithm picks a tile size of 36 for the L1 cache and 288 for the L2 cache.

Figure 7(a) shows performance results for a constant number of right-hand sides (`M` in Figure 2 is 100). 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 register tiling and instruction scheduling.

Figure 7(b) compares the performance of our code, tiled for two levels of the memory hierarchy, with code tiled for a single level with tile sizes ranging from 10 to 250 for an array size of 2000×2000 and $M = 100$. As can be seen, our two level scheme gives the best performance.

4.2 Cholesky Factorization

Cholesky factorization is used to solve symmetric positive-definite linear systems. Figure 8(a) shows one version of Cholesky factorization called the jki-$Cholesky$; 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 jki version shown here, the algorithm deduces that all 6 dimensions of the product space can be made fully permutable without reversal or skewing. It picks the following embeddings for the three statements:

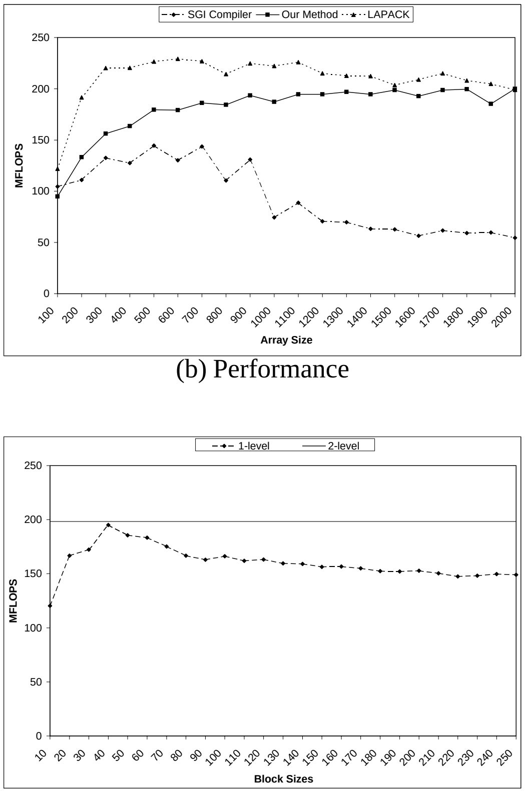
$$F_1(\begin{bmatrix} j \\ k \\ i \end{bmatrix}) = \begin{bmatrix} j \\ k \\ i \\ j \\ j \\ i \end{bmatrix} \qquad F_2(\begin{bmatrix} j \end{bmatrix}) = \begin{bmatrix} j \\ j \\ j \\ j \\ j \\ j \end{bmatrix}$$

$$F_3(\begin{bmatrix} j \\ i \end{bmatrix}) = \begin{bmatrix} j \\ j \\ i \\ j \\ j \\ i \end{bmatrix}$$

For these embeddings, the last three dimensions are redundant and are ignored. The remaining three dimensions are tiled. Our tile size selection algorithm chooses 36 for the L1 cache and 288 for the L2 cache. Figure 8(b) shows the result of tiling the three loops for varying matrix sizes. The code produced by our approach is many 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. The variation of performance with various tile sizes is shown in Figure 8(c) for an array size of 2000×2000.

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

(a) Original Code



(b) Performance

(c) Variation with tile size

Figure 8: Cholesky Factorization and its Performance

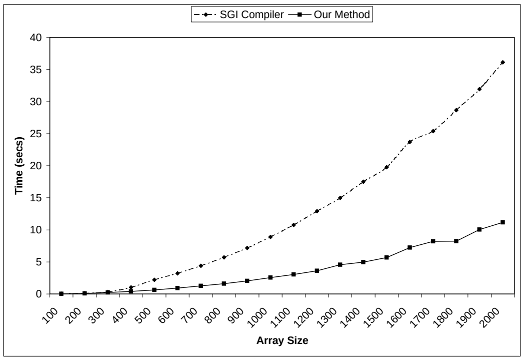
4.3 Jacobi

The Jacobi kernel is typical of code required to solve *pde*'s using 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 codes have imperfectly-nested loop nests. We show the results of applying our technique to the Jacobi kernel shown in Figure 9(a) which uses relaxation to solve Laplace's equation. It requires a non-trivial linear transformation of the product space.

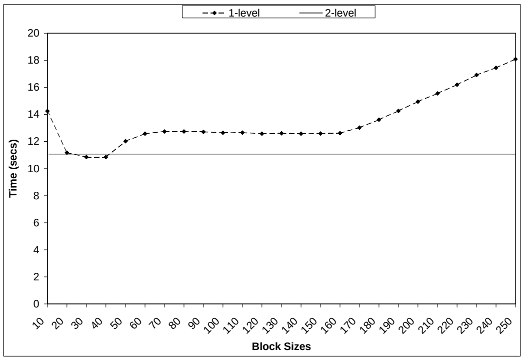
Our algorithm picks an embedding which corresponds intuitively to shifting the iterations of the two statements with respect to each other, and then fusing the resulting `i` and `j` loops. This not only permits tiling the loops but also exploits reuse between the two arrays in the two statements.

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

(a) Original Code



(b) Performance



(c) Variation with tile size

Figure 9: Jacobi and its Performance

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

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`. Our algorithm chooses a tile size of 20 for the L1 cache and 160 for the L2 cache.

Figure 9(b) shows the execution times for the code produced by our technique and by the SGI compiler for a fixed number of time-steps (100). Tiling the code improves performance significantly. Comparison with tiling with other tile sizes is shown in Figure 9(c).

4.4 Tomcatv

As a final example to demonstrate that our approach works on large codes, we consider the kernel from the tomcatv SPECfp benchmark suite.[7] 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. Our algorithm chooses to tile only for the L2 cache with a tiles-ize of 48. The results of applying the transformation are shown in Figure 10(a) for a fixed array size (253 from a reference input), and a varying number of time-steps. The line marked “Our Method” shows a performance improvement of around 18% over the original code. Additional improvement (line marked “Our Method (plus data transformation)”) can be obtained by doing a data transformation that transposes all the arrays as suggested in [24]. Comparison with other tile sizes is shown in Figure 10(b).

4.5 Discussion

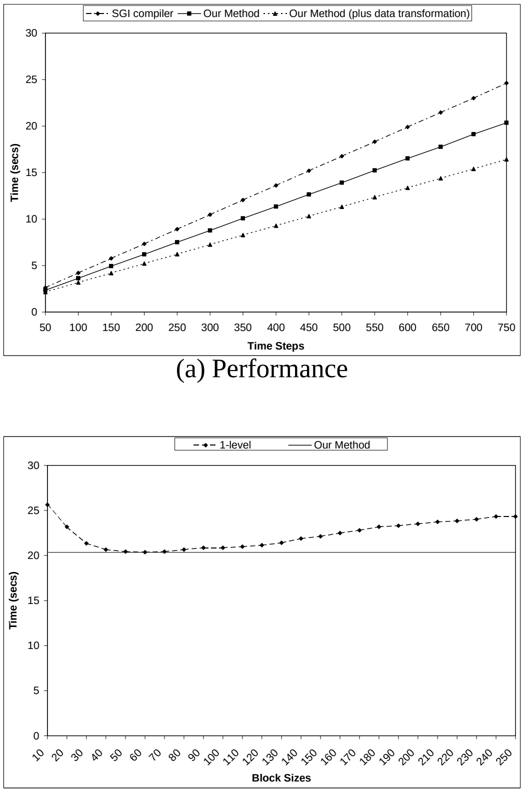
Our experiments show that tiling is important for the codes considered here. Although the SGI compiler is unable to tile these codes satisfactorily, our approach is able to produce code whose performance approaches that of hand-written libraries.

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. Our 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. Further, it allows us to pick good solutions by reducing the distance between dependent statement instances. It also does not require that programs conform to a specific structure.

Other kinds of embeddings have been used in the literature. For example, Feautrier [9] has solved schedul-

[7] Tomcatv is not directly amenable to our technique because it contains an exit test at the end of each time-step, so we consider the kernel without the exit condition. The resulting kernel can be tiled speculatively as demonstrated in [25].



(a) Performance

(b) Variation with tile size

Figure 10: Performance of tomcatv

ing 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 and Pugh [12] perform locality optimization by searching a space of pseudo-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. In their framework, they represent tiling by pseudo-affine mappings (using `mod` and `div`) but do not show how to obtain them. Lim and Lam [18] have used affine partitions to maximize parallelism. They derive constraints similar to our tiling constraints to parallelize programs with optimal synchronization. Unlike the previous two approaches, our approach prescribes a special space large enough to include all affine transformations and uses it to pick good solutions for tiling.

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.

Acknowledgements

We would like to thank the SC2000 referees and the Technical Papers Committee member Guang Gao for their valuable feedback which helped in improving the paper.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proc. International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.

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

[3] 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.

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

[5] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? In *INTEGRATION, the VLSI Journal*, volume 17, pages 33–51. 1994.

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

[7] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, June 1995.

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

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem - part 1: one dimensional time. *International Journal of Parallel Programming*, Oct. 1992.

[10] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 317–324, New York, July 1997. ACM Press.

[11] G. Golub and C. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[12] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, Feb. 1995.

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

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

[15] M. S. Lam, E. E. Rothberg, and M. 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, Apr. 8–11, 1991.

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

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

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

[19] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, pages 102–114, Aug. 1992.

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

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

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

[23] V. 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.

[24] Y. Song and Z. Li. A compiler framework for tiling imperfectly-nested loops. In *Proc. of 12th International Workshop on Languages and Compilers for Parallel Computing, (LCPC99)*, Aug. 1999.

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

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

[27] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29*, pages 274–286, Silicon Graphics, Mountain View, CA, 1996.

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

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