A Loop Transformation Theory and an Algorithm to Maximize Parallelism

Michael E. Wolf and Monica S. Lam, *Member, IEEE*

Abstract— **This paper proposes a new approach to transformations for general loop nests. In this approach, we unify all combinations of loop interchange, skewing and reversal as unimodular transformations. The use of matrices to model transformations has previously been applied only to those loop nests whose dependences can be summarized by distance vectors. Our technique is applicable to general loop nests where the dependences include both distances and directions.**

This theory provides the foundation for solving an open question in compilation for parallel machines: which loop transformations, and in what order, should be applied to achieve a particular goal, such as maximizing parallelism or data locality. This paper presents an efficient loop transformation algorithm based on this theory to maximize the degree of parallelism in a loop nest.

I. Introduction

LOOP transformations, such as loop interchange, reversal, skewing, and tiling (or blocking) [1], [4], [26] have been shown to be useful for two important goals: parallelism and efficient use of the memory hierarchy. Existing vectorizing and parallelizing compilers focused on the application of *individual* transformations on pairs of loops: when it is legal to apply a transformation, and if the transformation directly contributes to a particular goal. However, the more challenging problems of generating code for massively parallel machines, or improving data locality, require the application of compound transformations. It remains an open question as to how to combine these transformations to optimize general loop nests for a particular goal. This paper introduces a theory of loop transformations that offers an answer to this question. We will demonstrate the use of this theory with an efficient algorithm that maximizes the degrees of both fine- and coarse-grain parallelism in a set of loop nests via compound transformations.

Existing vectorizing and parallelizing compilers implement compound transformations as a series of pairwise transformations; for each step, the compiler must choose a transform that is legal and desirable to apply. A technique commonly used in today's parallelizing compilers is to decide *a priori* the order in which the compiler should attempt to apply transformations. This technique is inadequate because the choice and ordering of optimizations are highly program dependent, and the desirability of a transform cannot be evaluated locally, one step at a time. Another proposed technique is to "generate and test," that is, to explore all different possible combinations of transformations. This "generate and test" approach is expensive. Differently transformed versions of the same program may trivially have the same behavior and so need not be explored. For example, when vectorizing, the order of the outer loops is not significant. More importantly, generate and test approaches cannot search the entire space of transformations that have potentially infinite instantiations. Including loop skewing in a "generate and test" framework is thus problematic, because a wavefront can travel in an infinite number of different directions.

An alternative approach, based on matrix transformations, has been proposed and used for an important subset of loop nests. This class includes many important linear algebra codes on dense matrices; all systolic array algorithms belong to this class. They have the characteristic that their dependences can be represented by a set of integer vectors, known as *distance vectors*. Loop interchange, reversal, and skewing transformations are modeled as linear transformations in the iteration space [7], [8], [12], [13], [19], [21]. A compound transformation is just another linear transformation, being a product of several elementary transformations. This model makes it possible to determine the compound transformation directly in maximizing some objective function. Loop nests whose dependences can be represented as distance vectors have the property that an n-deep loop nest has at least $n-1$ degrees of parallelism [13], and can exploit data locality in all possible loop dimensions [24]. Distance vectors cannot represent the dependences of general loop nests, where two or more loops must execute sequentially. A commonly used notation for representing general dependences is *direction vectors* [2], [26].

This research combines the advantages of both approaches. We combine the mathematical rigor in the matrix transformation model with the generality of the vectorizing and concurrentizing compiler approach. We unify the various transformations, interchange or permutation, reversal and skewing as unimodular transformations, and our dependence vectors incorporate both distance and direction information. This unification provides a general condition to determine if the code obtained via a compound transformation is legal, as opposed to a specific legality test for each individual elementary transformation. Thus, the loop transformation problem can be formulated as directly solving for the unimodular transformation that maximizes some objective function, while satisfying a set of constraints. One important consequence of this approach

Manuscript received October 9, 1990; revised March 19, 1991. This work was supported in part by DARPA under Contract N00014-87-K-0828. A preliminary version of this paper appeared in The 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, CA, August 1990.

The authors are with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305.

IEEE Log Number 9100393.

is that code and loop bounds can be transformed once and for all, given the compound transformation.

Using this theory, we have developed algorithms for improving the parallelism and locality of a loop nest via loop transformations. Our parallelizing algorithm maximizes the degree of parallelism, the number of parallel loops, within a loop nest. By finding the maximum number of parallel loops, multiple consecutive loops can be coalesced to form a single loop with all the iterations; this facilitates load balancing and reduces synchronization overhead. Especially for loops with a small number of loop iterations, parallelizing only one loop may not fully exploit all the parallelism in the machine. The algorithm can generate coarse-grain and/or fine-grain parallelism; the former is useful in multiprocessor organizations and the latter is useful for vector machines and superscalar machines, machines that can execute multiple instructions per cycle. It can also generate code for machines that can use multiple levels of parallelism, such as a multiprocessor with vector nodes.

We have also applied our representation of transformations successfully to the problem of data locality. All modern machine organizations, including uniprocessors, employ a memory hierarchy to speed up data accesses; the memory hierarchy typically consists of registers, caches, primary memory, and secondary memory. As the processor speed improves and the gap between processor and memory speeds widens, data locality becomes more important. Even with very simple machine models (for example, uniprocessors with data caches), complex compound loop transformations may be necessary [9], [10], [17]. The consideration of data locality makes it more important to be able to combine primitive loop transformations in a systematic manner. Using the same theoretical framework presented in this paper, we have developed a locality optimization that applies compound unimodular loop transforms and tiling to use the memory hierarchy efficiently [24].

This paper introduces our model of loop dependences and transformations. We describe how the model facilitates the application of a compound transformation, using parallelism as our target. The model is important in that it enables the choice of an optimal transformation without an exhaustive search. The derivation of the optimal compound transformation consists of two steps. The first step puts the loops into a canonical form, and the second step tailors it to specific architectures. While the first step can be expensive in the worst case, we have developed an algorithm that is feasible in practice. We apply a cheaper technique to handle as many loops as possible, and use the more general and expensive technique only on the remaining loops. For most loop nests, the algorithm finds the optimal transformation in $O(n^3 d)$ time, where n is the depth of the loop nests and d is the number of dependence vectors. The second step of specializing the code for different granularities of parallelism is straightforward and efficient. After deciding on the compound transformation to apply, the code including the loop bounds is then modified. The loop transformation algorithm has been implemented in our SUIF (Stanford University Intermediate Format) parallelizing compiler. As we will show in the paper, the algorithms are simple, yet powerful.

This paper is organized as follows. We introduce the concept of our loop transformation model by first discussing the simpler program domain where all dependences are distance vectors. We first describe the model and present the parallelization algorithm. After extending the representation to directions, we describe the overall algorithm and the specific heuristics used. Finally we describe a method for rewriting the loop body and bounds after a compound transformation.

II. Loop Transformations on Distance Vectors

In this section, we introduce our basic approach to loop transformations by first studying the narrower domain of loops whose dependences can be represented as distance vectors. The approach has been adopted by numerous researchers particularly in the context of automatic systolic algorithm generation, a survey of which can be found in Ribas' dissertation [20].

A. Loop Nest Representation

In this model, a loop nest of depth n is represented as a finite convex polyhedron in the iteration space $\mathcal{Z}^n$ bounded by the loop bounds. Each iteration in the loop corresponds to a node in the polyhedron, and is identified by its index vector $\vec{p} = (p_1, p_2, \cdots, p_n)$;[1] p_i is the value of the ith loop index in the nest, counting from the outermost to innermost loop. In a sequential loop, the iterations are therefore executed in lexicographic order of their index vectors.

The execution order of the iterations is constrained by their data dependences. Scheduling constraints for many numerical codes are regular and can be succinctly represented by *dependence vectors*. A dependence vector in an n-nested loop is denoted by a vector $\vec{d} = (d_1, d_2, \cdots, d_n)$. The discussion in this section is limited to *distance vectors*, that is, $d_i \in \mathcal{Z}$. Techniques for extracting distance vectors are discussed in [12], [16], and [22]. General loop dependences may not be representable with a finite set of distance vectors; extensions to include directions are necessary and will be discussed in Section IV.

Each dependence vector defines a set of edges on pairs of nodes in the iteration space. Iteration $\vec{p_1}$ must execute before $\vec{p_2}$ if for some distance vector $\vec{d}$, $\vec{p_2} = \vec{p_1} + \vec{d}$. The dependence vectors define a partial order on the nodes in the iteration space, and any topological ordering on the graph is a legal execution order, as all dependences in the loop are satisfied. In a sequential loop nest, the nodes are executed in lexicographic order; thus, dependences extracted from such a source program can always be represented as a set of *lexicographically positive* vectors. A vector $\vec{d}$ is lexicographically positive, written $\vec{d} \succ \vec{0}$,

[1] The vector $\vec{p}$ is a column vector. In this paper, we sometimes represent column vectors as comma-separated tuples. That is,

$$(p_1, p_2, \cdots, p_n) = [p_1\ p_2\ \cdots\ p_n]^T = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix}.$$

if $\exists i : (d_i > 0$ and $\forall j < i : d_j \geq 0)$. ($\vec{0}$ denotes a zero vector, that is, a vector with all components equal to 0.)

Fig. 1(a) shows an example of a loop nest and its iteration space representation. Each axis represents a loop; each node represents an iteration that is executed within the loop nest. The 42 iterations of the loop are represented as a 6×7 rectangle in the two-dimensional space. Finally, each arrow represents a scheduling constraint. The access *a[I_2]* refers to data generated by the previous iteration from the same innermost loop, whereas the remaining two read accesses refer to data from the previous iteration of the outer loop. The dependence edges are all lexicographically positive: {(0, 1), (1, 0), (1, -1)}.

B. Unimodular Transformations

It is well known that loop transformations such as interchange, reversal, and skewing are useful in parallelization or improving the efficiency of the memory hierarchy. These loop transformations can be modeled as elementary matrix transformations; combinations of these transformations can simply be represented as products of the elementary transformation matrices. The optimization problem is thus to find the unimodular transformation that maximizes an objective function given a set of scheduling constraints.

We use the loop interchange transformation to illustrate the unimodular transformation model. A loop interchange transformation maps iteration (i, j) to iteration (j, i). In matrix notation, we can write this as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} j \\ i \end{bmatrix}.$$

The elementary permutation matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ thus performs the loop interchange transformation on the iteration space.

Since a unimodular matrix performs a linear transformation on the iteration space, $T\vec{p}_2 - T\vec{p}_1 = T(\vec{p}_2 - \vec{p}_1)$. Therefore, if $\vec{d}$ is a distance vector in the original iteration space, then $T\vec{d}$ is a distance vector in the transformed iteration space. Thus, in loop interchange, the dependence vector (d_1, d_2) is mapped into

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} d_2 \\ d_1 \end{bmatrix}$$

in the transformed space.

There are three elementary transformations:

- *Permutation:* A permutation σ on a loop nest transforms iteration $(p_1, \cdots, p_n)$ to $(p_{\sigma_1}, \cdots, p_{\sigma_n})$. This transformation can be expressed in matrix form as I_σ, the $n \times n$ identity matrix I with rows permuted by σ. The loop interchange above is an $n = 2$ example of the general permutation transformation.
- *Reversal:* Reversal of the ith loop is represented by the identity matrix, but with the ith diagonal element equal to -1 rather than 1. For example, the matrix representing loop reversal of the outermost loop of a two-deep loop nest is $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$.
- *Skewing:* Skewing loop I_j by an integer factor f with respect to loop I_i [26] maps iteration

  $$(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \cdots, p_{j-1}, p_j, p_{j+1}, \cdots, p_n)$$

  to

  $$(p_1, \cdots, p_{i-1}, p_i, p_{i+1}, \cdots, p_{j-1}, p_j + fp_i, p_{j+1}, \cdots, p_n).$$

  The transformation matrix T that produces skewing is the identity matrix, but with the element $t_{j,i}$ equal to f rather than zero. Since $i < j$, T must be lower triangular. For example, the transformation from Fig. 1(a) to Fig. 1(b) is a skew of the inner loop with respect to the outer loop by a factor of one, which can be represented as $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.

All these elementary transformation matrices are unimodular matrices [3]. A unimodular matrix has three important properties. First, it is square, meaning that it maps an n-dimensional iteration space to an n-dimensional iteration space. Second, it has all integral components, so it maps integer vectors to integer vectors. Third, the absolute value of its determinant is one. Because of these properties, the product of two unimodular matrices is unimodular, and the inverse of a unimodular matrix is unimodular, so that combinations of unimodular loop transformations and inverses of unimodular loop transformations are also unimodular loop transformations.

A compound transformation can be synthesized from a sequence of primitive transformations, and the effect of the transformation is represented by the products of the various transformation matrices for each primitive transformation.

C. Legality of Unimodular Transformations

We say that it is *legal* to apply a transformation to a loop nest if the transformed code can be executed sequentially, or in lexicographic order of the iteration space. We observe that if nodes in the transformed code are executed in lexicographic order, all data dependences are satisfied if the transformed dependence vectors are lexicographically positive. This observation leads to a general definition of a legal transformation and a theorem for legality of a unimodular transformation.

Definition 2.1: A loop transformation is *legal* if the transformed dependence vectors are all lexicographically positive.

Theorem 2.1: Let D be the set of distance vectors of a loop nest. A unimodular transformation T is legal if and only if $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$.

Using this theorem, we can evaluate if a compound transformation is legal directly. Consider the following example:

```
for I1 := 1 to N do
  for I2 := 1 to N do
    a[I1, I2] := f(a[I1, I2], a[I1 + 1, I2 − 1]);
```

This code has the dependence $(1, -1)$. The loop interchange transformation, represented by

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

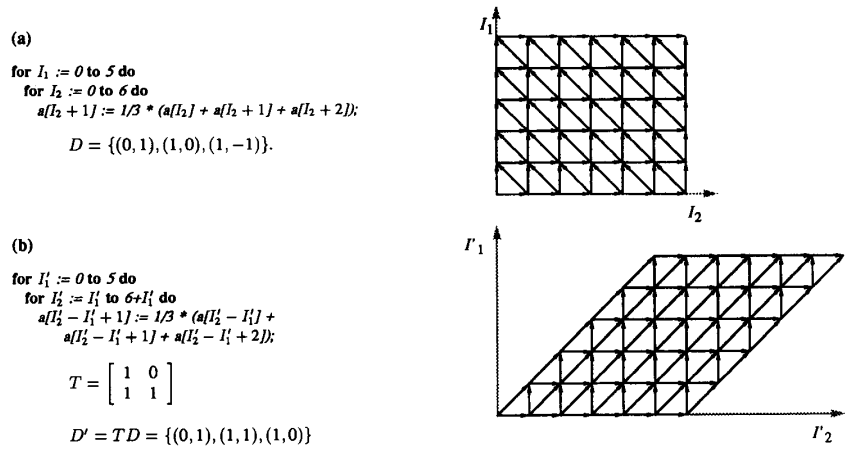


Fig. 1. Iteration space and dependences of (a) a source loop nest, and the (b) skewed loop nest.

is illegal, since $T(1,-1) = (-1,1)$ is lexicographically negative. However, compounding the interchange with a reversal, represented by the transformation

$$T' = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

is legal since $T'(1,-1) = (1,1)$ is lexicographically positive.

Similarly, Theorem 2.1 also helps to deduce the set of legal compound transformations. For example, if T is lower triangular with unit diagonals (loop skewing) then a legal loop nest will remain so after transformation. For another example, consider the dependences of the loop nest in Fig. 1(b). All the components of the dependences are nonnegative. This implies that any arbitrary loop permutation would render the transformed dependences lexicographically positive and is thus legal. We say such loop nests are *fully permutable*. Full permutability is an important property both for parallelization, discussed below, and for locality optimizations [24].

Theorem 2.2: Loops I_i through I_j of a legal computation with dependence vectors D are fully permutable if and only if

$$\forall \vec{d} \in D : \Big((d_1, \cdots, d_{i-1}) \succ \vec{0} \text{ or } (\forall i \le k \le j : d_k \ge 0)\Big).$$

III. Parallelizing Loops with Distance Vectors

We now study the application of the loop transformation theory to the problem of parallelization. The problem is to maximize the degree of parallelism, that is, the number of parallelizable loops. We are interested in running the code on machines supporting fine-grain parallelism, machines supporting coarse-grain parallelism and also machines supporting both levels of parallelism. We will show that n-deep loops whose dependences can be represented with distance vectors, have at least $n-1$ degrees of parallelism [13], exploitable at both fine and coarse granularity.

The algorithm consists of two steps: it first transforms the original loop nest into a canonical form, namely a fully permutable loop nest. It then transforms the fully permutable loop nest to exploit coarse and/or fine-grain parallelism according to the target architecture.

A. Canonical Form: A Fully Permutable Loop Nest

Loops with distance vectors have a special property that they can always be transformed into a fully permutable loop nest via skewing. It is easy to determine how much to skew an inner loop with respect to an outer to make those loops fully permutable. For example, if a doubly nested loop has dependences {(0, 1), (1, -2), (1, -1)}, then skewing the inner by a factor of two with respect to the outer produces {(0, 1), (1, 0), (1, 1)}. The following theorem explains how a legal loop nest can be made fully permutable.

Theorem 3.1: Let $L = \{I_1, \cdots, I_n\}$ be a loop nest with lexicographically positive distance vectors $\vec{d} \in D$. The loops in the loop nest can be made fully permutable by skewing.

Proof: We prove that if loops I_1 through I_i are fully permutable and all the dependences are lexicographically positive, then loop I_{i+1} can be skewed with respect to the outer loops to make all its dependence components nonnegative. Since skewing will not change the legality of the loop nest, this will legally include loop I_{i+1} in the fully permutable loop nest. Repeated application of this for $i = 1, \cdots, n-1$ makes the entire n-deep loop nest fully permutable, proving the theorem.

If $\forall \vec{d} \in D, d_{i+1} \ge 0$, then loop I_{i+1} is already permutable with loop I_i. Otherwise, since all dependences are lexicographically positive, $d_{i+1} < 0$ implies one of $d_1, \cdots, d_i$, say d_j, is positive. Therefore skewing loop I_{i+1} with respect to I_j by a factor

$$f \ge \max_{\{\vec{d} \mid \vec{d} \in D \wedge d_j \ne 0\}} \lceil -d_{i+1}/d_j \rceil$$

makes the transformed $i+1$st dependence component of the dependence vector nonnegative. By performing these skews until all $i+1$st dependence components are nonnegative, loop I_{i+1} can be incorporated into the fully permutable nest. □

B. Parallelization

Iterations of a loop can execute in parallel if and only if there are no dependences carried by that loop. This result is well known; such a loop is called a DOALL loop. To maximize the degree of parallelism is to transform the loop nest to maximize the number of DOALL loops. The following theorem rephrases the condition for parallelization in our notation:

Theorem 3.2: Let $(I_1, \cdots, I_n)$ be a loop nest with lexicographically positive dependences $\vec{d} \in D$. I_i is parallelizable if and only if $\forall \vec{d} \in D, (d_1, \cdots, d_{i-1}) \succ \vec{0}$ or $d_i = 0$.

Once the loops are made fully permutable, the steps to generate DOALL parallelism are simple. In the following, we first show that the loops in the canonical format can be trivially transformed to give the maximum degree of fine-grain parallelism. We then show how to generate the same degree of parallelism with coarser granularity. We then show how to obtain the same degree of parallelism with a lower synchronization cost, and how to produce both fine- and coarse-grain parallelism.

1) Finest Granularity of Parallelism: A nest of n fully permutable loops can be transformed to code containing at least $n-1$ degrees of parallelism [13]. In the degenerate case when no dependences are carried by these n loops, the degree of parallelism is n. Otherwise, $n-1$ parallel loops can be obtained by skewing the innermost loop in the fully permutable nest by each of the other loops and moving the innermost loop to the outermost position (Theorem B.2, see the Appendix). This transformation, which we call the *wavefront* transform, is represented by matrix

$$\begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}.$$

Fig. 2 shows the result of applying the wavefront transformation to the code in Fig. 1. The code is a result of first skewing the innermost loop to make the two-dimensional loop nest fully permutable, and applying the wavefront transformation to create one degree of parallelism. The figure also shows the transformed iteration space. We observe that there are no dependences between iterations within the innermost loop nest. We call this transform a wavefront transformation because it causes iterations along the diagonal of the original loop nest to execute in parallel. The wavefronts of the program in Fig. 1(b) are shown in Fig. 3.

This wavefront transformation automatically places the maximum DOALL loops in the innermost loops, maximizing fine-grain parallelism. This is the appropriate transformation for superscalar or VLIW (very long instruction word) machines. Although these machines have a low degree of parallelism, finding multiple parallelizable loops is still useful. Coalescing multiple DOALL loops prevents the pitfall of parallelizing only a loop with a small iteration count. It can reduce further the overhead of starting and finishing a parallel loop if code scheduling techniques such as software pipelining [15] are used.

2) Coarsest Granularity of Parallelism: For MIMD machines, having as many outermost DOALLs as possible reduces the synchronization overhead. The wavefront transformation produces the maximal degree of parallelism, but makes the outermost loop sequential if any are. For example, consider the following loop nest:

```
for I1 := 1 to N do
  for I2 := 1 to N do
    a[I1, I2] := f(a[I1 - 1, I2 - 1]);
```

This loop nest has the dependence (1, 1), so the outermost loop is sequential and the innermost loop is a doall. The wavefront transformation $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ does not change this. In contrast, the unimodular transformation $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$ transforms the dependence to (0, 1), making the outer loop a DOALL and the inner loop sequential. In this example, the dimensionality of the iteration space is two, but the dimensionality of the space spanned by the dependence vectors is only one. When the dependence vectors do not span the entire iteration space, it is possible to perform a transformation that makes outermost DOALL loops [14].

By choosing the transformation matrix T with first row $\vec{t}_1$ such that $\vec{t}_1 \cdot \vec{d} = 0$ for all dependence vectors $\vec{d}$, the transformation produces an outermost DOALL. In general, if the loop nest depth is n and the dimensionality of the space spanned by the dependences is n_d, it is possible to make the first $n - n_d$ rows of a transformation matrix T span the *orthogonal subspace* S of the space spanned by the dependences. This will produce the maximal number of outermost DOALL loops within the nest.

A vector $\vec{s}$ is in S if and only if $\forall \vec{d} : \vec{s} \cdot \vec{d} = 0$. S is the nullspace (kernel) of the matrix that has the dependences vectors as rows. Theorem B.4 shows how to construct a legal unimodular transformation T that makes the first $|S| = n - n_d$ rows span S. Thus, the outer $|S|$ loops after transformation are DOALLs. The remaining loops can be skewed to be fully permutable, and then wavefronted to get $n-1$ degrees of parallelism.

A practical though nonoptimal approach for making outer loops DOALL is simply to identify loops I_i such that all d_i are zero. Those loop can be made outermost DOALLs. The remaining loops in the tile can be wavefronted to get the remaining parallelism.

3) Tiling to Reduce Synchronization: It is possible to reduce the synchronization cost and improve the data locality of parallelized loops via an optimization known as *tiling* [25], [26]. Tiling is not a unimodular transformation. In general, tiling maps an n-deep loop nest into a $2n$-deep loop nest where the inner n loops include only a small fixed number of iterations. Fig. 4 shows the code after tiling the example in Fig. 1(b), using a tile size of 2×2. The two innermost loops execute the iterations within each tile, represented as 2×2 squares in the figure. The two outer loops, represented by the two axes in the figure, execute the 3×4 tiles. As the outer loops of the tiled code control the execution of the tiles, we will refer to them as the *controlling loops*.

```
for I'1 := 0 to 16 do
  doall I'2 := max(0,⌈(I'1 − 6)/2⌉) to min(5,⌊I'1/2⌋) do
    a[I'1 − 2I'2 + 1] := 1/3 * (a[I'1 − 2I'2] + a[I'1 − 2I'2 + 1]
      +a[I'1 − 2I'2 + 2]);
```

$$T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$$

$$D' = TD = \{(1,0),(2,1),(1,1)\}$$

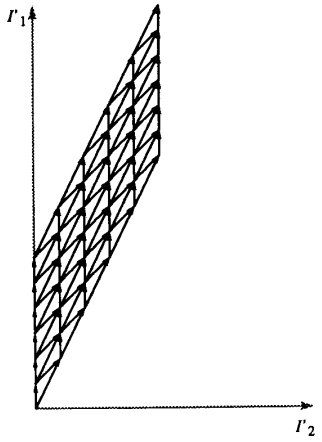


Fig. 2. Iteration space and dependences of Fig. 1(a) after skewing and applying the wavefront transformation.

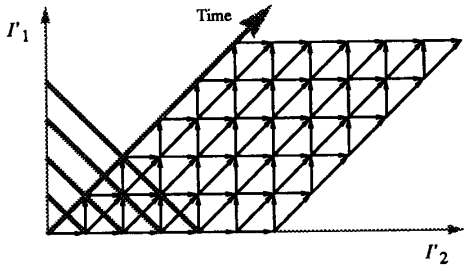


Fig. 3. Wavefronts of Fig. 1(b).

The same property that supports parallelization, full permutability, is also the key to tiling:

Theorem 3.3: Loops I_i through I_j of a legal computation can be tiled if the loops I_i through I_j are fully permutable.

Thus, loops in the canonical format of the parallelization algorithm can also be tiled. Moreover, the characteristics of the controlling loops resemble those of the original set of loops. An abstract view giving the dependences of the tiles in the above example is shown in Fig. 5. These controlling loops are themselves fully permutable and so easily parallelizable. However, each iteration of the outer n loops is a tile of iterations instead of an individual iteration. Tiling can therefore increase the granularity of synchronization [25] and data are often reused within a tile [24]. Without tiling, when a DOALL loop is nested within a non-DOALL loop, all processors must be synchronized at the end of each DOALL loop with a barrier. Using tiling, we can reduce the synchronization cost in the following two ways.

First, instead of applying the wavefront transformation to the loops in canonical form, we first tile the loops then apply the wavefront transformation to the controlling loops of the tiles. In this way, the synchronization cost is reduced by the size of the tile. An example of a wavefront of tiles is highlighted in Fig. 5.

To further reduce the synchronization cost, we can apply the concept of a DOACROSS loop to the tile level [25]. After tiling, instead of skewing the loops statically to form DOALL loops, the computation is allowed to skew dynamically by explicit synchronization between data dependent tiles. In the DOALL loop approach, tiles of each level must be completed before the processors may go on to the next, requiring a global barrier synchronization. In the DOACROSS model, each tile can potentially execute as soon as it is legal to do so. This ordering can be enforced by local synchronization. Furthermore, different parts of the wavefront may proceed at different rates as determined dynamically by the execution times of the different tiles. In contrast, the machine must wait for the slowest processor at every level with the DOALL method.

Tiling has two other advantages. First, within each tile, fine-grain parallelism can easily be obtained by skewing the loops *within* the tile and moving the DOALL loop innermost. In this way, we can obtain both coarse- and fine-grain parallelism. Second, tiling can improve data locality if there is data reuse across several loops [10], [24].

C. Summary

Using the loop transformation theory, we have shown a simple algorithm in exploiting both coarse- and fine-grain parallelism for loops with distance vectors. The algorithm consists of two steps: the first is to transform the code into the canonical form of a fully permutable loop nest. The second tailors the code to specific architectures via wavefront and tiling transformations.

IV. Direction Vectors

The dependences of general sequential loop nests cannot be represented as distances. Consider the following code:

```
for I1 := 0 to N do
  for I2 := 0 to N do
    b := g(b);
```

```
for II'1 := 0 to 5 by 2 do
  for II'2 := 0 to 11 by 2 do
    for I'1 := II'1 to min(5,II'1 + 1) do
      for I'2 := max(I'1, II'2) to min(6+I'1, II'2+1) do
        a[I'2 + 1] := 1/3 * (a[I'2] + a[I'2 + 1] + a[I'2 + 2]);
```

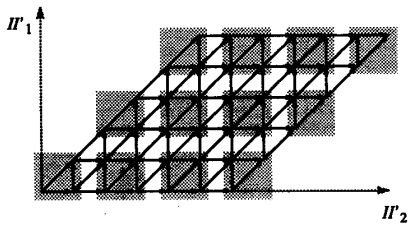


Fig. 4. Iteration space and dependences of tiled code from Fig. 1(b).

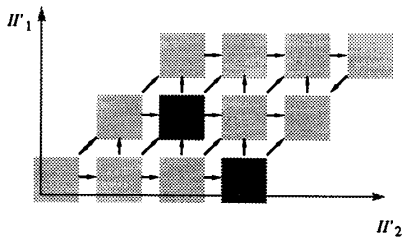


Fig. 5. Parallel execution of tiled loops.

Iteration (i, j) must precede iteration $(i, j+1)$, giving rise to a distance vector $(0, 1)$. But also, iteration (i, N) must precede iteration $(i+1, 0)$, giving rise to a $(1, -N)$ vector where N may not be known at compile time.

While the loop above does not have any exploitable parallelism, loops such as the following contain parallelism, but the dependences cannot be represented as distance vectors:

for $I_1 := \mathbf{0}$ **to** N **do**
  for $I_2 := \mathbf{0}$ **to** N **do**
    $a[I_1, I_2]$ **:=** $a[I_1 + 1, b[I_2]]$;

To represent this type of information, previous research on vectorizing and parallelizing compilers introduced the concept of *directions* [2], [26]. The dependence vector for the first example above would have been ('*', '*'), indicating that all the iterations are using the same data b, and must be serialized. In addition, the symbols '<' and '>' are used if the directions of the dependences can be determined: the second example would have a direction vector of ('<', '*').

We would like to extend the mathematical framework used on distance vectors to include directions. To do so, we make one key modification to the existing convention: legal "direction vectors" must also be "lexicographically positive." This modification makes it possible to use the same simple legality test for compound transformations involving direction vectors.

Each component d_i of a dependence vector $\vec{d}$ is now a possibly infinite range of integers, represented by $[d_i^{\min}, d_i^{\max}]$, where

$$d_i^{\min} \in \mathcal{Z} \cup \{-\infty\},\ d_i^{\max} \in \mathcal{Z} \cup \{\infty\} \text{ and } d_i^{\min} \leq d_i^{\max}.$$

A single dependence vector therefore represents a set of distance vectors, called its *distance vector set*:

$$\mathcal{E}(\vec{d}) = \{(e_1, \cdots, e_n) | e_i \in \mathcal{Z} \text{ and } d_i^{\min} \leq e_i \leq d_i^{\max}\}$$

The dependence vector $\vec{d}$ is also a distance vector if each of its components is a degenerate range containing a singleton value, meaning $d_i^{\min} = d_i^{\max}$. We use the notation '+' as shorthand for $[1, \infty]$, '−' as shorthand for $[-\infty, -1]$, and '±' as shorthand for $[-\infty, \infty]$. They correspond to the directions '<', '>', and '*', respectively.

All the properties discussed in Sections II and III hold for the distance vector sets of the dependence vectors. For example, a unimodular transformation is legal if $\forall \vec{d} : \forall \vec{e} \in \mathcal{E}(\vec{d}) : T\vec{e} \succ \vec{0}$. Instead of handling infinite distance vector sets, we define an arithmetic on vectors with possibly infinite integer ranges, so we can operate on dependence vectors directly.

We say that a component d is *positive*, written $d > 0$, if its minimum $d^{\min}$ is positive; d is *nonnegative*, written $d \geq 0$, if its minimum is nonnegative. Likewise, d is *negative* or *nonpositive* if its maximum $d^{\max}$ is negative or nonpositive, respectively. Therefore, $d \not> 0$ is not equivalent to $d \leq 0$. With these definitions of component comparison, the definition used in lexicographically positive distance vectors also applies to general dependence vectors: $\vec{d} \succ \vec{0}$, if $\exists i : (d_i > 0$ and $\forall j < i : d_j \geq 0)$. A dependence vector $\vec{d}$ is lexicographically nonnegative if it is lexicographically positive or all its components have $d^{\min} = 0$. This definition implies that if a dependence vector is lexicographically positive, then all the distance vectors in its distance vector set also are.

Since we require all dependences to be lexicographically positive, we do not allow the dependences ('±', '±'), or ('*', '*'), as arose in a previous example. The dependences for such sequential loops are further refined and represented as $(0, '+'), ('+', '\pm')$.

To enable calculation of $T\vec{d}$, where $\vec{d}$ is a vector of components and T is a matrix of integers, we define component addition to be

$$[a, b] + [c, d] = [a + c, b + d]$$

where for all $s \in \mathcal{Z} \cup \{\infty\}$, $s + \infty$ is ∞ and for all $s \in \mathcal{Z} \cup \{-\infty\}$, $s + -\infty$ is $-\infty$. Thus $2 + [-3, \infty] = [2, 2] + [-3, \infty] = [-1, \infty]$. Likewise, we define multiplication of a component by a scalar as

$$s[a, b] = \begin{cases} [sa, sb], & \text{if } s \geq 0 \\ [sb, sa], & \text{otherwise} \end{cases}$$

where $s \cdot \infty$ is ∞ for positive s, 0 if s is 0, and $-\infty$ for negative s, and likewise for a factor times $-\infty$. Component subtraction $a-b$ is defined to be $a+(-1)b$. These definitions of addition, subtraction, and multiplication are conservative in that

$$\vec{e}_1 \in \mathcal{E}(\vec{d}_1) \text{ and } \vec{e}_2 \in \mathcal{E}(\vec{d}_2) \Rightarrow f(\vec{e}_1, \vec{e}_2) \in \mathcal{E}(f(\vec{d}_1, \vec{d}_2))$$

where f is a function that performs a combination of the defined operations on its operands. The converse

$$\vec{e} \in \mathcal{E}(f(\vec{d}_1, \vec{d}_2)) \Rightarrow$$
$$\left(\exists \vec{e}_1 \in \mathcal{E}(\vec{d}_1) \text{ and } \exists \vec{e}_2 \in \mathcal{E}(\vec{d}_2) : f(\vec{e}_1, \vec{e}_2) = \vec{e}\right)$$

is not necessarily true.

For an example, with transformation $T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ and $\vec{d} = (0, '+')$, the precise distance vector set of $T\vec{d}$ is {(1, 1), (2, 2), (3, 3), ...}. Using the component arithmetic defined above, the resulting dependence vector is ('+', '+'), which represents the ranges of each component correctly and is the best representation of the infinite set given the dependence representation. The choice of the direction representation is a compromise; it is powerful enough to represent most common cases without the cost associated with the more complex and precise representation; but the application of certain transformations may result in loss of information. This drawback to direction vectors can be overcome by representing them as distance vectors whenever possible, since no information is lost during distance vector manipulation. In the above example, we can represent the dependence by (0, 1) rather than by $(0, '+')$, since by transitivity both are identical. The transformed dependence is then (1, 1), which exactly summarizes the dependence after transformation, rather than ('+', '+'), which is imprecise.

When direction vectors are necessary, information about the dependences may be lost whenever skewing is applied. We can only guarantee that $\mathcal{E}\left(T^{-1}(T\vec{d})\right) \supset \mathcal{E}\left(\vec{d}\right)$, but $T^{-1}(T\vec{d}) = \vec{d}$ is not necessarily true. Thus, for general dependence vectors, the analogue of Theorem 2.1 is a pair of more restrictive theorems.

Theorem 4.1: Let D be the set of dependence vectors of a computation. A unimodular transformation T is legal if $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$.

Theorem 4.2: Let D be the set of dependence vectors of a computation. A unimodular transformation T that is a product of only permutation and reversal matrices is legal if and only if $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$.

V. Parallelization with Distances and Directions

Recall that an n-deep loop nest has at least $n-1$ degrees of parallelism if its dependences are all distance vectors. In the presence of direction vectors, not all loops can be made fully permutable, and such degrees of parallelism may not be available. The parallelization problem therefore is to find the maximum number of parallelizable loops.

The parallelization algorithm again consists of two steps: the first to transform the loops into the canonical form, and the second tailors it to fine- and/or coarse-grain parallelism. Instead of a single fully permutable loop nest, the canonical format for loops with general dependences is a *nest* of fully permutable loop nests with each outer fully permutable loop nest as large as possible. As we will show below, once the loops are placed in this canonical format, we can maximize the total degree of parallelism by maximizing the degree of parallelism for each fully permutable loop nest.

A. An Example with Direction Vectors

Let us first illustrate the procedure with an example.

```
for I1 := 1 to N do
  for I2 := 1 to N do
    for I3 := 1 to N do
      (a[I1, I3],b[I1, I2, I3]) :=
f(a[I1, I3], a[I1 + 1, I3 - 1],b[I1, I2, I3], b[I1, I2, I3 - 1]);
```

The loop body above is represented by an $N \times N \times N$ iteration space. The references $a[I_1, I_3]$ and $a[I_1+1, I_3-1]$ give rise to a dependence of $(1, '\pm', -1)$. The references $a[I_1, I_3]$ and $a[I_1, I_3]$ do not give rise to a dependence of $(0, '\pm', 0)$, which is not lexicographically positive, but rather to a dependence of $(0, '+', 0)$. $b[I_1, I_2, I_3]$ and $b[I_1, I_2, I_3]$ give rise to a dependence of (0, 0, 0), which we ignore since it is not a cross-iteration edge. Finally, $b[I_1, I_2, I_3]$ and $b[I_1, I_2, I_3 - 1]$ give rise to a dependence of (0, 0, 1). The dependence vectors for this nest are

$$D = \{(0, '+', 0), (1, '\pm', -1), (0, 0, 1)\}.$$

None of the three loops in the source program can be parallelized as it stands; however, there is one degree of parallelism that can be exploited at either a coarse- or fine-grain level.

By permuting loops I_2 and I_3 loops, and skewing the new middle loop with respect to the outer by a factor of 1, the algorithm transforms the code to canonical form:

```
for I1' := 1 to N do
  for I2' := I1' + 1 to I1' + N do
    for I3' := 1 to N do
      (a[I1', I2' - I1'],b[I1', I3', I2' - I1']) :=
                f(a[I1', I2' - I1'], a[I1' + 1, I2' - I1' - 1],
                b[I1', I3', I2' - I1'], b[I1', I3', I2' - I1' - 1]);
```

The transformation matrix T and the transformed dependences D' are

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

and

$$D' = \{(0, 0, '+'), (1, 0, '\pm'), (0, 1, 0)\}.$$

The transformation is legal since the resulting dependences are all lexicographically positive. The resulting two outermost loops form one set of fully permutable loop nests, since interchanging these loops leaves the dependences lexicographically positive. The last loop is in a fully permutable loop by itself.

The two-loop fully permutable set can be transformed to provide one level of parallelism by application of the wavefront skew. The transformation matrix T for this phase of transformation, and the transformed dependences D'' are

$$T' = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and

$$D'' = \{(0,0,\text{'+'}),(1,1,\text{'}\pm\text{'}),(1,0,0)\}$$

and the transformed code is

for $I_1'' := 3$ **to** $3N$ **do**
  doall $I_2'' := \max(1,\lceil (I_1'' - N)/2 \rceil)$ **to**
    $\min(N,\lfloor (I_1'' - 1)/2 \rfloor)$ **do**
    for $I_3'' := 1$ **to** N **do**
      $(a[I_2'', I_1'' - 2I_2''], b[I_2'', I_3'', I_1'' - 2I_2'']) :=$
        $f(a[I_2'', I_1'' - 2I_2''],\ a[I_2'' + 1, I_1'' - 2I_2'' - 1],$
        $b[I_2'', I_3'', I_1'' - 2I_2''],\ b[I_2'', I_3'', I_1'' - 2I_2'' - 1]);$

Applying this wavefront transformation to all the fully permutable loop nests will produce a loop nest with the maximum degree of parallelism. For some machines, tiling may be preferable to wavefronting, as discussed in Section III-B3.

To implement this technique, there is no need to generate code for the intermediate step. The transformation to get directly from the example to the final code is simply

$$T'' = T'T = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

B. Canonical Form: Fully Permutable Nests

The parallelization algorithm attempts to create the largest possible fully permutable loop nests, starting from the outermost loops. It first applies unimodular transformations to move loops into the outermost fully permutable loop nest, if dependences allow. Once no more loops can be placed in the outermost fully permutable loop nest, it recursively places the remaining loops into fully permutable nests further in. The code for this algorithm, called *Make_Nests*, is presented in Appendix A. It calls *Find_FP_Nest* to choose the fully permutable nest to place outermost at each step in the algorithm; that algorithm is discussed in Section VI.

For loops whose dependences are distance vectors, Section III shows that their parallelism is easily accessible once they are made into a fully permutable nest. Similarly, once a loop nest with general dependence vectors is put into a nest of largest fully permutable nests, its parallelism is readily extracted. This canonical form has three important properties that simplify its computation and make its parallelism easily exploitable:

1) The largest, outermost fully permutable nest is unique in the loops it contains, and is a superset of all possible outermost permutable loop nests (Theorem B.5). We need to consider only combinations of loops that make up the *largest* outermost fully permutable nest.
2) If a legal transformation exists for a loop nest originally, then there exists a *legal* transformation that will generate code with the largest possible outermost fully permutable loop nest placed outermost (Theorem B.3). This property makes it safe to use a greedy algorithm that constructs fully permutable loop nests incrementally starting with the outermost fully permutable loop nest and working inwards.
3) Finally, a greedy algorithm that places the largest outermost fully permutable loop nest, and then recursively calls itself on the remaining loops to place loops further in, exposes the maximum degree of parallelism possible via unimodular transformation (Theorem B.6). The maximum degree of parallelism in a loop nest is simply the sum of the maximal degree of parallelism of all the fully permutable nests in canonical form.

C. Fine- and Coarse-Grain Parallelism

To obtain the maximum finest granularity of parallelism, we perform a wavefront skew on all the fully permutable nests, and permute them so that all the DOALL loops are innermost, which is always legal.

To obtain the coarsest granularity of parallelism, we simply transform each fully permutable nest to give the coarsest granularity of parallelism, as discussed in Section III-B2. If the fully permutable nest contains direction vectors, the technique of Section III-B2 must be modified slightly.

Let S be the orthogonal subspace of the dependences, as in Section III-B2. Although the dependences are possibly directions, we can still calculate this space precisely. Since the nest is fully permutable, $d_k^{\min} \geq 0$. We can assume without loss of generality that $d_k^{\max} = d_k^{\min}$ or $d_k^{\max} = \infty$ by enumerating all finite ranges of components. If some component of $\vec{d}$ is unbounded, that is $d_k^{\max} = \infty$, then our requirement on $\vec{s} \in S$ that $\vec{s} \cdot \vec{d} = 0$, implies that $s_k = 0$. This puts the same constraint on S as if there had been a distance vector $(0,\ldots,0,1,0,\ldots,0)$, with a single 1 in the kth entry. Thus, we can calculate S by finding the nullspace of the matrix with rows consisting of $(d_1^{\min},\cdots,d_n^{\min})$ and of the appropriate $(0,\ldots,0,1,0,\ldots,0)$ vectors.

These rows are a superset of the dependences within the fully permutable loop nest, but they are all lexicographically positive and all their elements are nonnegative. Thus, we can use these as the distance vectors of the nest and apply the technique of Section III-B2 for choosing a legal transformation T to apply to make the first $|S|$ rows of the transformation matrix span S, and the technique will succeed. Since S is calculated precisely and the first $|S|$ rows of T span S, the result is optimal.

VI. Finding Fully Permutable Loop Nests

In this section, we discuss the algorithm for finding the largest outermost fully permutable loop nest; this is the key step in parallelization of a general loop nest. Finding the largest fully permutable loop nest in the general case is

expensive. A related but simpler problem, referred to here as the time cone problem, is to find a linear transformation matrix for a *finite* set of distance vectors such that the first components of all transformed distance vectors are all positive [12], [13], [18], [21]. This means that the rest of the loops are DOALL loops. We have shown that if all the distance vectors are lexicographically positive, this can be achieved by first skewing to make the loops fully permutable, and wavefronting to generate the DOALL loops. However, if the distances are not lexicographically positive, the complexity of typical methods to find such a transformation assuming one exists is at least $O(d^{n-1})$, where d is the number of dependences and n is the loop nest depth [21]. The problem of finding the largest fully permutable loop nest is even harder, since we need to find the largest nest for which such a transformation exists.

Our approach is to use a cheaper technique on as many loops as possible, and apply the expensive time cone technique only to the few remaining loops. We classify loops into three categories:

1) serializing loops, loops with dependence components including both $+\infty$ and $-\infty$; these loop cannot be included in the outermost fully permutable nest and can be ignored for that nest.
2) loops that can be included via the SRP transformation, an efficient transformation that combines permutation, reversal, and skewing.
3) the remaining loops; they may possibly be included via a general transformation using the time cone method.

Appendix A contains the implementation of *Find_FP_Nest*, the algorithm that finds an outermost fully permutable loop nest. The parameters to *Find_FP_Nest* are the set of loops and the set of dependences D not satisfied by a loop enclosing the current loops. The routine returns the loops that were not placed in the nest, the transformed dependences, and the loops in this nest. In addition, the actual implementation must keep track of exactly which transformations are being implicitly performed by the algorithm in order to update the loop body and loop bounds. The algorithm first removes the loops in category 1 from consideration, as discussed below, and then calls the SRP transformation to handle loops of category 2. If necessary, it then calls *General_Transform* to handle loops of category 3.

As we expect the loop depth to be low to begin with and further reduced since most loops will typically fall in categories 1 and 2, we recommend only a two-dimensional time cone method for loops of category 3. If there are two or fewer loops in category 3, the algorithm is optimal, maximizing the degree of parallelism. Otherwise, the two-dimensional time cone method can be used heuristically to find a good transformation. The algorithm can transform the code optimally in the common cases. For the algorithm to fail to maximize the degree of parallelism, the loop has to be at least five deep. The overall complexity of the algorithm is $O(n^3 d)$.

A. Serializing Loops

As proven below, a loop with both positive and negative infinite components cannot be included in the outermost fully permutable nest. Consequently, the loop must go in a fully permutable nest further in, and cannot be run in parallel with other loops in the outermost fully permutable nest. We call such a loop a *serializing* loop.

Theorem 6.1: If loop I_k has dependences such that $\exists \vec{d} \in D : d_k^{\min} = -\infty$ and $\exists \vec{d} \in D : d_k^{\max} = \infty$ then the outermost fully permutable nest consists only of a combination of loops not including I_k.

Proof: $\exists \vec{d} \in D : d_k^{\min} = -\infty$ implies that the transformation T that produces m outermost fully permutable loop nests must be such that $\forall i \leq m : t_{i,k} \leq 0$. Likewise, $\exists \vec{d} \in D : d_k^{\max} = \infty$ implies that $\forall i \leq m : t_{i,k} \geq 0$. Thus if $\exists \vec{d} \in D : (d_k^{\min} = -\infty) \wedge \exists \vec{d} \in D : (d_k^{\max} = \infty)$ then it must be the case that $\forall i \leq m : t_{i,k} = 0$. Thus, none of the m outermost loops after transformation contains a component of I_k. □

In our example in Section V, the middle loop is serializing, so cannot be in the outermost fully permutable loop nest. Once the other two loops are placed outermost, the only dependence from D' that is not already made lexicographically positive by the first two loops is $(0, 0, '+')$, so that for the next fully permutable nest, there were no more serializing loops that had to be removed from consideration.

When a loop is removed from consideration, the resulting dependences are no longer necessarily lexicographically positive. For example, suppose a loop nest has dependences

$$\{(1, '\pm', 0, 0), (0, 1, 2, -1), (0, 1, -1, 1)\}.$$

The second loop is serializing, so we only need to consider placing in the outermost nest the first, third and fourth loops. If we just examine the dependences for those loops {(1, 0, 0), (0, 2, -1), (0, -1, 1)}, we see that some dependences become lexicographically negative.

B. The SRP Transformation

Now that we have eliminated serializing loops from consideration for the outermost fully permutable nest, we apply the SRP transformation. SRP is an extension of the skewing transformation we used in Section III to make a nest with lexicographically positive distances fully permutable. This technique is simple and effective in practice. The loop in Section V is an example of code that is amenable to this technique.

We first consider the effectiveness of skewing when applied to loops with general dependence vectors. First, there may not be a legal transformation that can make two loops with general dependences fully permutable. For example, no skewing factor that can make the components of vector $(1, '-')$ all nonnegative. Moreover, the set of dependences vectors under consideration may not even be lexicographically positive, as discussed above. Thus, even if the dependences are all distance vectors, it is not necessarily the case that $n - 1$ parallel loops can be extracted. For example, suppose a subset of two loops in a loop nest has dependence vectors {(1, -1), (-1, 1)}. These dependences are anti-parallel, so no row of any matrix T can have a nonnegative dot product with both vectors, unless the row itself is the zero vector, which would make the matrix singular. Nonetheless, there are many common cases

for which a simple reverse and skew transformation can extract parallelism from the code.

Theorem 6.2: Let $L = \{I_1, \cdots, I_n\}$ be a loop nest with lexicographically positive dependences $\vec{d} \in D$, and $D^i = \{\vec{d} \in D | (d_1, \cdots, d_{i-1}) \not\succ \vec{0}\}$. Loop I_j can be made into a fully permutable nest with loop I_i, where $i < j$, via reversal and/or skewing, if

$$\forall \vec{d} \in D^i : (d_j^{\min} \neq -\infty \wedge (d_j^{\min} < 0 \rightarrow d_i^{\min} > 0)) \text{ or}$$
$$\forall \vec{d} \in D^i : (d_j^{\max} \neq \infty \wedge (d_j^{\max} > 0 \rightarrow d_i^{\min} > 0)) .$$

Proof: All dependence vectors for which $(d_1, \cdots, d_{i-1}) \succ \vec{0}$ do not prevent loops I_i and I_j from being fully permutable and can be ignored. If

$$\forall \vec{d} \in D^i : (d_j^{\min} \neq -\infty \wedge (d_j^{\min} < 0 \rightarrow d_i^{\min} > 0))$$

then we can skew loop I_j by a factor of f with respect to loop I_i where

$$f \geq \max_{\{\vec{d} | \vec{d} \in D^i \wedge d_i^{\min} \neq 0\}} \lceil -d_j^{\min} / d_i^{\min} \rceil$$

to make loop I_j fully permutable with loop I_i. If instead the condition

$$\forall \vec{d} \in D^i : (d_j^{\max} \neq \infty \wedge (d_j^{\max} > 0 \rightarrow d_i^{\min} > 0)) .$$

holds, then we can reverse loop I_j and proceed as above. □

From this theorem, we derive the *Find_Skew* algorithm in the Appendix. It takes as input D, the set of dependences that has not been satisfied by loops outside this loop nest. It also takes as input the loop nest N and the loop I_j. It attempts to skew loop I_j with respect to the fully permutable loop nest N so that all dependences (that are not satisfied by outer loops) will have nonnegative d_j components. It returns whether it was able to successfully skew, as well as the skews performed and the new dependence vectors. The run time for this algorithm is $O(|N|d + d)$, where d is the number of dependences in the loop nest. If N is empty, then this routine returns successfully if loop I_j was such that all $d_j \geq 0$, or if loop I_j could be reversed to make that so.

The SRP transformation has several important properties, proofs of which can be found in [23]. First, although the entire transformation is constructed as a series of a permutation, reversal, and skew combination, the complete transformation can be expressed as the application of one permutation transformation, followed by a reversal of zero or more of the loops, followed by a skew. That is, we can write the transformation T as $T = SRP$, where P is a permutation matrix, R is a diagonal matrix with each diagonal entry being either 1 or -1 (loop reversal), and S is a lower triangular matrix with ones on the diagonal (loop skewing). Note that while T produces a legal fully permutable loop nest, just applying P may not produce a legal loop nest.

Second, as an extension to the skewing transform presented in Section III, SRP converts a loop nest with only lexicographically positive distance vectors into a single fully permutable loop nest. Thus, SRP finds the maximum degree of parallelism for loops in this special case. Generalizing this observation, if there exists an SRP transformation that makes dependence vectors lexicographically positive, and none of the transformed dependence vectors has an unboundedly negative component ($\forall \vec{d} : d^{\min} \neq -\infty$), our algorithm again will place all the loops into a single fully permutable loop nest.

Third, the *Find_FP_Nest* algorithm that uses SRP to find successive fully permutable loop nests will always produce a legal order for the loops if all the dependences were initially lexicographically positive, or could have been made so by some SRP transformation.

Fourth, the complexity of the SRP transformation algorithm is $O(m^2 d)$ where m is the number of loops considered. If *Find_FP_Nest* does not attempt transformations on loops in category 3, then it is $O(n^2 d)$ where n is the number of loops in the entire loop nest.

Fifth, if *Find_FP_Nest* with SRP fails to create any fully permutable nest with more than one loop, then there is no unimodular transformation that could have done so. To rephrase slightly, *Find_FP_Nest* with SRP produces DOALL parallelism whenever any is available via a unimodular transformation, even if it finds less than the optimal degree of parallelism.

C. General Two-Dimensional Transformations

In general, the SRP transformation alone cannot find all possible parallelism. Consider again the example of a loop with dependences $D = \{(1, '\pm', 0, 0), (0, 1, 2, -1), (0, 1, -1, 1)\}$. The first loop can obviously be placed in the outermost fully permutable nest. The second loop is serializing, and cannot be placed in the outermost fully permutable loop nest. The question is whether the third and fourth loops can be placed into the outermost fully permutable nest. That is, does a transformation exist that makes all components of dependences $\{(1, 0, 0), (0, 2, -1), (0, -1, 1)\}$ nonnegative after transformation. While SRP will not succeed, such a transformation does indeed exist. The first loop can go in the fully permutable nest; the challenge is to find a T such that $T(2, -1)$ and $T(-1, 1)$ are fully permutable. Finding such a T is a reformulation of the time cone problem discussed above. We present an efficient $O(d)$ algorithm to solve for a transformation in the special case when there are exactly two loops in the nest.

The algorithm consists of two steps. The first is to find a direction $\vec{t}$ in the iteration space such that $\forall \vec{d} \in D : \vec{t} \cdot \vec{d} > 0$. The second is to find a unimodular matrix T that contains $\vec{t}$ in the first row, so that $\vec{t}$ is the direction of the outermost loop after transformation.

To find the "time" direction $\vec{t}$, we note that each dependence will restrict the legal space of $\vec{t}$ to a cone in the iteration space. We illustrate this method using the example of subvectors $(2, -1)$ and $(-1, 1)$ in Fig. 6. As shown in the figure, each dependence restricts the legal $\vec{t}$ to a half-plane that has a nonnegative dot product with $\vec{t}$. The intersection of the legal times form a cone, any of which is a legal outer loop. If the cone is empty, then there is no possible transformation that achieves our goal. Otherwise, the extreme edges of the cone can be (vector) summed to produce a direction $\vec{t}$.[2] The

[2] Banerjee [5] discusses a method for choosing the direction that maximizes the number of iterations that can be performed in parallel.

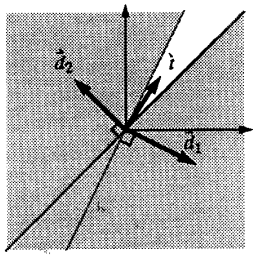


Fig. 6. An example of a two-dimensional time-cone.

components of the vector $\vec{t}$ are chosen such that their greatest common divisor is one. In our example, $(t_1, t_2) = (2, 3)$. The technique can easily be extended to any number of distance and/or direction vectors.

The transformation matrix T will transform the dependences in the desired manner if

$$T = \begin{bmatrix} t_1 & t_2 \\ x & y \end{bmatrix}$$

and T is unimodular. To achieve this, we choose x and y integral such that the determinant of T is unity, which occurs when $t_1 y - t_2 x = 1$. Since t_1 and t_2 are relatively prime, an x and y pair are easily found using the Extended Euclidean algorithm. For our example, Euclid's algorithm produces $x = 1$ and $y = 2$, and we get

$$T = \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \text{ and } T^{-1} = \begin{bmatrix} 2 & -3 \\ -1 & 2 \end{bmatrix}.$$

The (I_1, I_2) iteration is transformed by T into the (I_1', I_2') iteration. By the construction of the transformation T, the I_1' component of all dependences are positive, so all dependences are lexicographically positive. Thus, we can skew the I_2' loop with respect to the I_1' loop by calling SRP to make the nest fully permutable.

By incorporating the two-dimensional transform into *Find_FP_Nests*, the execution time of the complete parallelizing algorithm becomes potentially $O(n^3 d)$. In the worst case, SRP takes $O(n^2 d)$ but places no loops in the nest, and then *Find_FP_Nest* searches $\binom{n}{2}$ pairs of loops to find a pair for which the two-dimensional time cone method succeeds. Thus, at worst it takes $O(n^2 d)$ to place two loops into the nest, which makes the worst case run time of the algorithm $O(n^3 d)$. In the case that SRP was sufficient to find the maximum degree of parallelism, the two-dimensional transformation portion of the algorithm is never executed, and the run time of the parallelization algorithm remains $O(n^2 d)$.

VII. Implementating the Transformations

In this section, we discuss how to express a loop transformed by unimodular transformation and tiling back as executable code. In either case, the problem consists of two parts: rewriting the body of the loop nest and rewriting the loop bounds. It is easy to rewrite the body of the loop nest; the difficult part is the loop bounds. We first discuss unimodular transformations, then tiling.

A. Unimodular Transformations

Suppose we have the loop nest

for $I_1 := \cdots$
$\cdots$
for $I_n := \cdots$
$S(I_1, \cdots, I_n)$;

to which we apply the unimodular transformation T. The transformation of the body S only requires that I_j be replaced by the appropriate linear combination of I''s, where the I''s are the indexes for the transformed loop nest:

$$\begin{bmatrix} I_1 \\ \vdots \\ I_n \end{bmatrix} = T^{-1} \begin{bmatrix} I_1' \\ \vdots \\ I_n' \end{bmatrix}.$$

Performing this substitution is all that is required for the loop body. The remainder of this section discusses the transformation on the loop bounds.

1) Scope of Allowable Loop Bounds: Our method of determining the loop bounds after unimodular transformation requires that the loop bounds be of the form

for $I_i := \max(L_i^1, L_i^2, \ldots)$ **to** $\min(U_i^1, U_i^2, \ldots)$ **by** 1 **do**

where

$$L_i^j = \left\lceil \left(l_{i,0}^j + l_{i,1}^j I_1 + \ldots + l_{i,i-1}^j I_{i-1} \right) / l_{i,i}^j \right\rceil$$
$$\text{and } U_i^j = \left\lfloor \left(u_{i,0}^j + u_{i,1}^j I_1 + \ldots + u_{i,i-1}^j I_{i-1} \right) / u_{i,i}^j \right\rfloor$$

and all $l_{i,k}^j$ and $u_{i,k}^j$ are known constants, except possibly for $l_{i,0}^j$ and $u_{i,0}^j$, which must still be invariant in the loop nest. (If a ceiling occurs where we need a floor it is a simple matter to adjust $l_{i,0}^j$ and $u_{i,0}^j$ and replace the ceiling with the floor, and likewise if a floor occurs where we need a ceiling.) If any loop increments are not one, then they must first be made so, for example via loop normalization [1]. If the bounds are not of the proper form, then the given loop cannot be involved in any transformations, and the loop nest is effectively divided into two: those outside the loop and those nested in the loop.

It is important to be able to handle loop bounds of this complexity. Although programmers do not often write bound expressions with minima, maxima, and integer divisions, loop transformations can create them. In particular, we consider tiling as separate from unimodular transformation, so in the case of double level blocking, we can transform the nest with a unimodular matrix, tile, and then further transform the tiled loops and tile again inside. Since tiling can create loops of this complexity, we wish to be able to handle them.

2) Transforming the Loop Bounds: We outline our method for determining the bounds of a loop nest after transformation by unimodular matrix T. We explain the general method and demonstrate it by permuting the loop nest in Fig. 7 to make I_3 the outermost loop and I_1 the innermost loop, i.e., by applying the transformation $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$.

An example loop nest:

for $I_1 := 1$ **to** n_1 **do**
  for $I_2 := 2I_1$ **to** n_2 **do**
    for $I_3 := 2I_1 + I_2 - 1$ **to** $\min(I_2, n_3)$ **do**
      $S(I_1, I_2, I_3)$;

Step 1: Extract inequalities:

$$\begin{array}{lll} I_1 \geq 1 & & I_1 \leq n_1 \\ I_2 \geq 2I_1 & & I_2 \leq n_2 \\ I_3 \geq 2I_1 + I_2 - 1 & I_3 \leq I_2 & I_3 \leq n_3 \end{array}$$

Step 2: Find absolute maximum and minimum for each loop index:

$$\begin{array}{ll} I_1^{\min} = 1 & I_1^{\max} = n_1 \\ I_2^{\min} = 2 \times 1 = 2 & I_2^{\max} = n_2 \\ I_3^{\min} = 2 \times 1 + 2 - 1 = 3 & I_3^{\max} = \min(n_2, n_3) \end{array}$$

Step 3: Transform indices: $I_1 \Rightarrow I_3'$ $I_2 \Rightarrow I_2'$ $I_3 \Rightarrow I_1'$

Inequalities			Maxima and minima	
$I_3' \geq 1$	$I_3' \leq n_1$		$I_1'^{\min} = 3$	$I_1'^{\max} = \min(n_2, n_3)$
$I_2' \geq 2I_3'$	$I_2' \leq n_2$		$I_2'^{\min} = 2$	$I_2'^{\max} = n_2$
$I_1' \geq 2I_3' + I_2' - 1$	$I_1' \leq I_2'$	$I_1' \leq n_3$	$I_3'^{\min} = 1$	$I_3'^{\max} = n_1$

Step 4: Calculate new loop bounds:

Index	Inequality (index on LHS)	Substituting in $I'^{\min}$ and $I'^{\max}$	Result
I_2'	$I_2' \geq 2I_3'$	$I_2' \geq 2$	$I_2' \geq 2$
	$I_2' \geq I_1'$		$I_2' \geq I_1'$
	$I_2' \leq n_2$		$I_2' \leq n_2$
	$I_2' \leq I_1' - 2I_3' + 1$	$I_2' \leq I_1' - 1$	$I_2' \leq I_1' - 1$
I_3'	$I_3' \geq 1$		$I_3' \geq 1$
	$I_3' \leq n_1$		$I_3' \leq n_1$
	$I_3' \leq (I_1' - I_2' + 1)/2$		$I_3' \leq \lfloor (I_1' - I_2' - 1)/2 \rfloor$

The loop nest after transformation:

for $I_1' := 3$ **to** $\min(n_3, n_2)$ **do**
  for $I_2' := I_1'$ **to** $\min(n_2, I_1' - 1)$ **do**
    for $I_3' := 1$ **to** $\min(n_1, \lfloor (I_1' - I_2' + 1)/2 \rfloor)$ **do**
      $S(I_3', I_2', I_1')$;

Fig. 7. Finding the loop bounds after unimodular transformation.

Step 1: Extract inequalities. The first step is to extract all inequalities from the loop nest. Using the notation from Section VII-A1, the bounds can be expressed as a series of inequalities of the form $I_i \geq L_i^j$ and $I_i \leq U_i^j$.

Step 2: Find the absolute maximum and minimum for each loop index. The second step is to use the inequalities to find the maximum and minimum possible values of each loop index. This can be easily done from the outermost to the innermost loop by substituting the maxima and minima outside the current loop of interest into the bound expressions for that loop. That is, since the lower bound of I_i is $\max_j(L_i^j)$, the smallest possible value for I_i is the maximum of the smallest possible values of L_i^j.

We use $I_i^{\min}$ and $L_i^{j,\min}$ to denote the smallest possible value for I_i and L_i^j, and $I_k^{\min_{i,j}}$ to denote either $I_k^{\min}$ or $I_k^{\max}$, whichever minimizes $(l_{i,k}^j I_k^{\min_{i,j}})/l_{i,i}^j$:

$$I_i^{\min} = \max_j(L_i^{j,\min})$$

where

$$L_i^{j,\min} = \left\lceil \left(l_{i,0}^j + \sum_{k=1}^{i-1} l_{i,k}^j I_k^{\min_{i,j}} \right) / l_{i,i}^j \right\rceil$$

and

$$I_k^{\min_{i,j}} = \begin{cases} I_k^{\min}, & \text{sign}(l_{i,k}^j) = \text{sign}(l_{i,i}^j) \\ I_k^{\max}, & \text{otherwise.} \end{cases}$$

Similar formulas hold for $U_i^{j,\min}$, $L_i^{j,\max}$, and $U_i^{j,\max}$.

Step 3: Transform indexes. We now use $(I_1, \cdots, I_n) = T^{-1}(I_1', \ldots, I_n')$ to replace the I_j's variables in the inequalities by I_j''s. The maximum and minimum for a given I_j' is easily calculated. For example, if $I_1' = I_1 - 2I_2$, then $I_1'^{\max} = I_1^{\max} - 2I_2^{\min}$ and $I_1'^{\min} = I_1^{\min} - 2I_2^{\max}$. In general, we use $(I_1', \cdots, I_n') = T(I_1, \cdots, I_n)$ to express the I_j''s in terms of the I_j's. If $I_j' = \sum a_j I_j$, then

$$I_k'^{\min} = \sum_{k=1}^{n} a_j \begin{cases} I_j^{\min} & a_j > 0 \\ I_j^{\max} & \text{otherwise} \end{cases}$$

and likewise for $I_k'^{\max}$.

Step 4: Calculate new loop bounds. The lower and upper bounds for loop I_1' are just $I_1'^{\min}$ and $I_1'^{\max}$. Otherwise, to determine the loop bounds for loop index I_i', we first rewrite each inequality from step three containing I_i' by solving for I_i', producing a series of inequalities of the form $I_i' \leq f(\ldots)$ and $I_i' \geq f(\ldots)$. Each inequality of the form $I_i' \leq f(\ldots)$ contributes to the upper bound. If there is more than one such

expression, then the minimum of the expressions is the upper bound. Likewise, each inequality of the form $I'_i \geq f(\ldots)$ contributes to the lower bound, and the maximum of the right-hand sides is taken if there is more than one.

An inequality of the form $I'_i \geq f(I'_j)$ contributes to the lower bound of loop index I'_i. If $j < i$, meaning that loop I'_j is outside loop I'_i , then the expression does not need to be changed since the loop bound of I'_i can be a function of the outer index I'_j. If loop I'_j is nested within loop I'_i, then the bound for loop index I'_i cannot be a function of loop index I'_j. In this case we must replace I'_i by its minimum or maximum, whichever minimizes f. A similar procedure is applied to the upper loop bounds. The loop bounds produced as a result of these manipulations again belong to the class discussed in Section VII-A1, so that our methods can calculate the loop bounds after further transformation of these loops.

3) Discussion of Method: The algorithm to determining loop bounds is efficient. For an n-deep loop nest, there are at least $2n$ inequalities, since each loop has at least one lower and upper bound. In general, each loop bound may be maximum and minimum functions, with every term contributing one inequality. Let the number of inequalities be q. Both steps 1 and 2 in the algorithm are linear. The third step requires a matrix–vector multiply and a scan through the inequalities, and so is $O(n^2 + q)$. The fourth step is potentially the most expensive, since all q inequalities could potentially involve all n loop indexes, for a total cost of $O(nq)$. Thus, the loop bound finding algorithm is $O(nq)$.

The resulting loop nest, while correct, may contain excessive maxima and minima computation in the loop bounds, and may contain outer loop iterations that have no loop body iterations executed. Neither of these problems occurs in the body of the innermost loop, and thus the extra computation cost should be negligible. These problems can be removed via well-known integer linear system algorithms [11].

B. Tiling Transformation

The technique we apply for tiling does not require any changes to the loop body. We thus discuss only the changes to the loop nest and bounds. While it has been suggested that strip-mining and interchanging be applied to determine the bounds of a tiled loop, this approach is not straightforward when the loop bounds are not rectangular [26]. A more direct method is as follows. When tiling, we partition the iteration space, whatever the shape of the bounds, as in Fig. 8. Each rectangle represents a computation performed by a tile; some tiles may contain little or even no work.

Tiling a nest $(I_i, \cdots, I_j)$ adds $j - i + 1$ *controlling loops*, denoted by $(II_i, \cdots, II_j)$, to the loop nest. The lower bound on the I_k is now the maximum of the original lower bound and II_k; similarly, the upper bound is the minimum of the original upper bound and $II_k + S_k - 1$, where S_k is the size of the tile in the k loop. For loop II_k, the lower and upper bounds are simply the absolute minimum and maximum values of the original I_k, and the step size is S_k. As shown in Fig. 8, some of these tiles may be empty. The time wasted in determining that the tile is empty should be negligible when compared to the execution of the large number of nonempty tiles in the loop. Once again, these problems can be removed via well-known integer linear system algorithms [11].

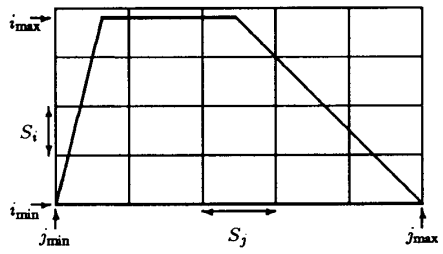


Fig. 8. Tiling a trapezoidal loop (2-D).

Applying these methods to the permuted example loop nest from Fig. 7, we can tile to get the following:

for $II'_1 := 3$ **to** $\min(N_2, N_3)$ **by** S_1 **do**
for $II'_2 := 2$ **to** N_2 **by** S_2 **do**
for $II'_3 := 1$ **to** N_1 **by** S_3 **do**
  for $I'_1 := \max(3, II'_1)$ **to** $\min(N_3, N_2, II'_1 + S_1 - 1)$ **do**
  for $I'_2 := \max(I'_1, II'_2)$ **to** $\min(N_2, I'_1 - 3, II'_2 + S_2 - 1)$ **do**
  for $I'_3 := \max(1, II'_3)$ **to** $\min(N_1, \lfloor (I'_1 - I'_2 - 1)/2 \rfloor, II'_3 + S_3 - 1)$ **do**
    $S(I'_3, I'_2, I'_1)$;

Note that I'_1, I'_2, and I'_3 are fully permutable, and so are the II'_1, II'_2, and II'_3. Each of these two groups can further be transformed to deliver parallelism at the fine and/or coarse granularity of parallelism, as discussed in Section III-B3. The same loop bound conversion algorithm described here is sufficient to support such further transforms, because the loop bounds of each group again satisfy the general format allowed.

VIII. Conclusions

This paper proposes a new approach to transformations for general loop nests with distance and direction dependence vectors. We unify the handling of distance and direction vectors by representing direction vectors as an infinite set of distance vectors. In this approach, dependence vectors represent precedence constraints on the iterations of a loop. Therefore, dependences extracted from a loop nest must be lexicographically positive. This departure from previous research on vectorizing and parallelizing compilers leads to a simple test for legality of compound transformations: any code transformation that leaves the dependences lexicographically positive is legal.

This model unifies all combinations of loop interchange, reversal and skewing as unimodular transformations. The use of matrices to model transformations has previously been applied only to special cases where dependences can be summarized by distance vectors. We extended vector arithmetic to include directions in a way such that the important and common operations on these vectors are efficient. With this model, we can easily determine a compound transform to be legal: a transform is legal if it yields lexicographically

positive dependence vectors when multiplied to the original dependence vectors. The ability to directly relate a compound transform to the characteristics of the final code allows us to prune the search for the optimal transform. A problem such as maximizing parallelism is now simply one of finding the unimodular transformation matrix that maximizes an objective function, subject to the legality constraints.

To demonstrate the usefulness of this model, this paper applies this loop transformation theory to the problem of maximizing the degree of coarse- or fine-grain parallelism in a loop nest. There are three main results. First, the maximum degree of parallelism can be achieved by transforming the loops into a nest of coarsest fully permutable loop nests, and wavefronting the fully permutable nests. Second, this canonical form of coarsest fully permutable nests can be transformed mechanically to yield maximum degrees of coarse- and/or fine-grain parallelism. Third, while it is expensive to find the coarsest fully permutable loop nest, our efficient heuristics can find the maximum degrees of parallelism for loops whose nesting level is less than five.

This loop transformation theory has both theoretical and practical benefits. This matrix transformation model makes it possible to formulate various loop transformation problems as mathematical optimization problems, prove certain transforms to be optimal in an infinite space of general loop transforms, and to devise efficient and effective heuristics to complex problems. Besides applying the theory to maximizing parallelism, we have successfully used this model in developing an efficient algorithm that applies unimodular and tiling transforms to improve the data locality of loop nests [24]. The elegance of this theory helps reduce the complexity of the implementation. Once the dependences are extracted, the derivation of the compound transform simply consists of matrix and vector operations. Once the transformation is determined, a straightforward algorithm applies the transformation to the loop bounds and derives the final code.

Appendix A
The Parallelization Algorithms

```
algorithm Make_FP_Nests
        (D: set of dependence,
        n: integer)
    return
        (success: boolean,
        D: set of dependence,
        T: matrix,
        fp_nests: list of index);

  fp_nests: list of index := [];
  bot: integer := 0;
  T: matrix := identity;
  while bot < n do
    T_f: matrix;
    top: integer := bot+1;
    (D,T_f,bot) := Find_FP_Nest(D,top,n);
    if top > bot then
      return (false,D,T,fp_nests);
    end if;
    fp_nests := fp_nests+[top];
    T := T_f T;
  end while;
  return (true,D,T,fp_nests);
end algorithm;
```

```
algorithm Find_FP_Nest
        (D: set of dependence,
        top: index, /* first unplaced */
        n: index)
    return
        (D: set of dependence,
        T: matrix,
        bot: index); /* last placed */

  bot: index;
  T,T_g: matrix;
  D^top: set of dependence := {d⃗ ∈ D|(d_1,...,d_{top-1}) ≯ 0⃗};
  I: set of index := {i ∈ {top,...,n}|
          ∀d⃗ ∈ D^top : d_i^min ≠ −∞ or
          ∀d⃗ ∈ D^top : d_i^max ≠ ∞};
  (D,T,bot,I) := SRP(D,top,top−1,I);
  if I ≠ ∅ then
    (D,T_g,bot,I) := General_Transform(D,top,bot,I);
    T := T_g T;
  end if;
  return (D,T,bot);
end algorithm;
```

```
algorithm SRP
        (D: set of dependence,
        top: index, /* top of fp nest */
        cur: index, /* last loop in fp nest so far */
        I: set of index)
    return
        D: set of dependence,
        T: matrix,
        bot: index,
        I: set of index); /* available loops */

  done: boolean := false;
  T,T_s,T_p: matrix := identity;
  while not done do
    skewed: boolean;
    done := true;
    foreach c ∈ I repeat
      (skewed,D,T_s) := Find_Skew(D,c,top,cur);
      if skewed then
        done := false;
        T_p := identity with rows c and cur+1 swapped;
```

```
      T := T_p T_s T;
      D := ∀d⃗ ∈ D : swap components d_{cur+1} and d_c;
      I := I − {c}; /* c now placed */
      if cur+1 ∈ I then
        I := I − {cur+1} + {c};
      end if;
      cur := cur+1;
    end if;
  end foreach;
 end while;
 return (D,T,cur,I);
end algorithm;
```

```
algorithm Find_Skew
        (D: set of dependence,
        j: index, /* candidate for skewing */
        top,cur: index) /* fp nest top to cur */
    return
        (success: boolean,
        D: set of dependence,
        T: matrix);

  N: set of index := {top,...,cur};
  T: matrix := zero;
  D^top: set of dependence := {d⃗ ∈ D|(d_1,...,d_{top−1}) ≯ 0⃗};

  /* pass one: determine whether skewing possible */
  if ∀d⃗ ∈ D^top : d_j^min ≠ −∞ and
        (d_j^min ≥ 0 or ∃k ∈ N : d_k^min > 0) then
    T_{j,j} := 1;
  else if ∀d⃗ ∈ D^top : d_j^max ≠ ∞ and
        (d_j^max ≤ 0 or ∃k ∈ N : d_k^min > 0) then
    T_{j,j} := −1;
    foreach d⃗ ∈ D do d_j := −d_j; end foreach;
  else
    return (false,D,T);
  end if;

  /* pass two: determine skews */
  foreach d⃗ ∈ D^top do
    if d_j^min < 0 then
      choose any k : k ∈ N ∧ d_k^min > 0;
      T_{j,k} := max(T_{j,k}, ⌈−d_j^min/d_k^min⌉);
    end if;
  end foreach;

  /* pass three: fix dependences */
  foreach d⃗ ∈ D do
    d_j := d_j + Σ_{k∈N} T_{j,k} d_k;
  end foreach;
  return (true,D,T);
end algorithm;
```

```
algorithm General_Transform
        (D: set of dependence,
        top: index, /* top of current nest */
        cur: index, /* 1..cur already ordered */
        I: set of index)
    return
        (D: set of dependence,
        T: matrix,
        bot: index,
        I: set of index)

  success: boolean := false;
  c1,c2: index;
  T_tmp,T: matrix := identity;
  foreach c1, c2 ∈ I, c1 ≠ c2 do
    (success,D,T_tmp) := Find_2d_Trans(D,c1,c2,cur);
    if success then
      T := T_tmp;
      cur := cur+2;
      if I ≠ ∅ then
        (D,T_tmp,cur,I) := SRP(D,top,cur,I);
        T := T_tmp T;
        if I ≠ ∅ then
          (D,T_tmp,cur,I) := General_Transform(D,top,cur,I);
          T := T_tmp T;
        end if;
      end if;
      return (D,T,cur,I);
    end if;
  end foreach;
  return (D,T,cur,I);
end algorithm;
```

```
algorithm Find_2d_Trans
        (D: set of dependence,
        c1,c2: index, /* loops to transform */
        cur: index, /* 1..cur already ordered */
        I: set of index)
    return
        (success: boolean, /* true if 2D transf. exists */
        D: set of dependence, /* the new dependences */
        T: matrix, /* the transf. found */
        I: set of index) /* unplaced loops */

  /* See Section VI-C */
end algorithm;
```

APPENDIX B
PROOFS

We first state and prove a lemma about unimodular matrices that we need for the proofs in this section.

Lemma B.1: If $\gcd(t_{1,1}, t_{1,2}, \cdots, t_{1,n}) = 1$, then an $n \times n$ unimodular matrix T exists with $(t_{1,1}, t_{1,2}, \cdots, t_{1,n})$ as its first row.

Proof: This lemma is true for $n = 2$, as we saw in Section VI. We prove that it is true for general n by mathematical induction.

If $t_{1,i} = 0$ for $i = 2, \cdots, n$, then $t_{1,1}$ is 1 or -1 and a solution is the identity matrix with the first diagonal element replaced by $t_{1,1}$. So assume at least one of $t_{1,i}$ is nonzero in $i = 2, \cdots, n$. Let $g = \gcd(t_{1,2}, \cdots, t_{1,n})$. We construct the matrix

$$U = \begin{bmatrix} t_{1,1} & t_{1,2} & \cdots & t_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & & \vdots \\ 0 & u_{n-1,2} & \cdots & u_{n-1,n} \\ x & t_{1,2}s/g & \cdots & t_{1,n}s/g \end{bmatrix}$$

where rows 2 through $n-1$ of U are chosen so that the upper right $n-1 \times n-1$ submatrix has a determinant of $\pm g$. This is possible because a unimodular matrix exists with a first row of $(t_{1,2}/g, \cdots, t_{1,n}/g)$, by the induction hypothesis. Given this construction, the lower right $n-1 \times n-1$ submatrix of U has a determinant of $\pm s$. The determinant of U is therefore $(t_{1,1})(\pm s) - (x)(\pm g)$ which can be written as the determinant of one of the four matrices represented by $\begin{bmatrix} t_{1,1} & \pm g \\ x & \pm s \end{bmatrix}$. The top row has a gcd of unity, so that x and s can be chosen to make the appropriate matrix unimodular. Thus, we have constructed an $n \times n$ unimodular matrix with the required first row. □

A. Parallelism in a Fully Permutable Nest

Theorem B.2: Applying the wavefront transformation

$$\begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

to a fully permutable loop nest maximizes the degree of parallelism within the nest. For an n-dimensional loop nest, the maximal degree of parallelism is $n-1$ if there are any dependences carried within the loop nest, and n otherwise.

Proof: Suppose the fully permutable loop nest is loops I_i through I_j. Let $D^i = \{\vec{d} \in D | (d_1, \cdots, d_{i-1}) \not\succ \vec{0}\}$. The dependences in D^i are the only ones of interest, since all others are automatically satisfied. For all dependences $\vec{d} \in D^i$, the loops are fully permutable means that $d_k \geq 0$ for all $k = i, \cdots, j$. Thus, for all $\vec{e} \in \mathcal{E}(\vec{d})$ either $e_i + \cdots + e_j > 0$ or all $e_k = 0$. In either case, the resulting $i+1$st through jth loops are DOALL loops. It is also easy to verify all dependences that were lexicographically positive before are still so. Thus, the transformation produces a legal ordering of the original nest, as well as producing $j - i$ DOALL loops. It is not possible to have all loops be DOALLs unless there are no dependences in D^i, in which case the above transformation also produces all DOALL loops. □

B. Legality of Making Loops Outermost

Theorem B.3: Suppose there exists a unimodular transformation T such that $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$. Let the vector $\vec{u}$ be such that $\gcd(u_1, \cdots, u_n) = 1$ and $\forall \vec{d} \in D : \vec{u} \cdot \vec{d} \geq 0$. Then there exists a unimodular transformation U with first row $\vec{u}$ such that $\forall \vec{d} \in D : U\vec{d} \succ \vec{0}$.

Proof: We first prove the theorem under the assumption that all the dependence vectors $\vec{d}$ are distance vectors $\vec{e}$.

Let $\vec{t_i}$ be the ith row of T, and define the constants $a_1, \cdots, a_m$ so that $\vec{u} = a_1\vec{t_1} + \cdots + a_m\vec{t_m}$, where $a_m \neq 0$. We prove the theorem by constructing a sequence of $n \times n$ unimodular matrices $U_1, \cdots, U_m$. Each U_j we construct will have the property that $U_j\vec{e} \succeq_j \vec{0}$, where $\succeq_j \vec{0}$ means that the first j components are lexicographically nonnegative. U_j is of the form

$$U_j = \begin{bmatrix} A_j & 0 \\ 0 & I_{n-m} \end{bmatrix} T$$

where each A_j is an $m \times m$ unimodular matrix and I_{n-m} is the $n-m \times n-m$ identity matrix. That is, the first m rows of U_j are combinations of the first m rows of T and the last $n-m$ rows of U_j are identical to the last $n-m$ rows of T. We will show that U_m meets the requirements for U in the theorem statement.

We construct A_1 by choosing as its first row $(a_1, \cdots, a_m)$. Such an A_1 must exist by Lemma B.1. Thus, the first row of U_1 is therefore $\vec{u}$, and $\vec{u} \cdot \vec{e}$ is given to be nonnegative for all $\vec{e}$. Thus, U_1 satisfies the property that $U_1\vec{e} \succeq_1 \vec{0}$.

We now construct A_2 such that $U_2\vec{e} \succeq_2 \vec{0}$. We define the constants c_i to be such that $\vec{t_1} = c_1\vec{u}_{1,1} + \cdots + c_m\vec{u}_{1,m}$, where $\vec{u}_{1,k}$ is the kth row of U_1. If $m > 1$, then c_2 through c_m cannot all be zero, so we can define c_i' such that $c_i' = c_i / \gcd(c_2, \cdots, c_m)$. Then we define the matrix A_2 by

$$A_2 = \begin{bmatrix} 1 & 0 & 0 & \cdots \\ 0 & c_2' & c_3' & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} A_1$$

where rows 3 through m of the premultiplying matrix are chosen to make it unimodular, as is possible from Lemma B.1. U_2 again has a first row of $\vec{u}$. Now we show that for all $\vec{e}$, $U_2\vec{e} \succeq_2 \vec{0}$. The first component of $U_2\vec{e}$ is $\vec{u} \cdot \vec{e}$, which is given to be nonnegative. If it is positive, then the first two components of $U_2\vec{e}$ are trivially lexicographically positive; if it is zero, we much check for the nonnegativity of the second component of $U_2\vec{e}$. The second row of U_2 is a linear combination of $\vec{u}$ and $\vec{t_1}$ only. In fact, the second row of U_2 is exactly $(\vec{t_1} - c_1\vec{u}) / \gcd(c_2, \cdots, c_m)$. When the first component of $U_2\vec{e}$ is zero, then $\vec{u} \cdot \vec{e}$ is zero and the second component of $U_2\vec{e}$ is a positive multiple of $\vec{t_1} \cdot \vec{e}$. But $\vec{t_1} \cdot \vec{e}$ is nonnegative because T is a legal transformation. Thus, the first two entries of $U_2\vec{e}$ must form a lexicographically nonnegative vector.

We repeat this process, premultiplying A_{j-1} to construct a new matrix A_j with the property that for all $\vec{e}$, $U_j\vec{e} \succeq_j \vec{0}$. We achieve this by making the jth row of U_j be a linear combination of $\vec{t}_{j-1}$ and the first $j-1$ rows of U_{j-1}. As long as $j \leq m$, this construction is always possible. This process terminates with the construction of U_m.

We now show that U_m meets the requirements for U in the problem statement. U_m is unimodular and the first row of U_m is $\vec{u}$. To complete the proof, we must show that for all $\vec{e}$, $U_m\vec{e} \succ \vec{0}$. U_m is constructed so that $U_m\vec{e} \succeq_m \vec{0}$. If $U_m\vec{e} \succ_m \vec{0}$, then $U_m\vec{e} \succ \vec{0}$. The only other possibility is that the first m components of $U_m\vec{e}$ are all zero. In that case, it must be that the first m components of $T\vec{e}$ are also all zero, since the first m rows of U_m are just a combination of the first m rows of T. Since the first m components of $U_m\vec{e}$ and $T\vec{e}$ are identical, and U_m and T share the same last $n-m$ rows, it must be that $U_m\vec{e} = T\vec{e}$. Since $T\vec{e}$ is lexicographically positive, this completes the proof for sets of distance vectors.

Before extending the proof to direction vectors, we first outline the difficulty of doing so. Suppose we have the transformation $U = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}$ and the dependence vector $\vec{d} = \begin{bmatrix} [1,\infty] \\ [0,\infty] \end{bmatrix}$. $U\vec{d} = \begin{bmatrix} [0,\infty] \\ [-\infty,\infty] \end{bmatrix}$ so the transformation appears to be illegal, even though in fact for each $\vec{e} \in \mathcal{E}(\vec{d})$, $U\vec{e}$ is lexicographically positive. This happens because although the first component of the result vector may be zero, or the second component may be negative, both of these conditions cannot happen in the same resulting $U\vec{e}$. In constructing a U with first row (0, 1) as above, we may get the U above, which satisfies all the distances, but does not have the property that $U\vec{d} \succ \vec{0}$. However, the transformation $U' = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ has the same first row and is legal both in the sense that $\forall \vec{e} \in \mathcal{E}(\vec{d}) : U'\vec{e} \succ \vec{0}$ and $U'\vec{d} \succ \vec{0}$. To prove the theorem for general dependence vectors, we must prove that given a set of direction vectors, and a first row $\vec{u}$ such that $\forall \vec{d} \in D : \vec{u} \cdot \vec{d} \geq 0$, we can find a U' with a first row of $\vec{u}$ such that $\forall \vec{d} \in D : U'\vec{d} \succ \vec{0}$.

We begin with a U for which all the distance vectors in each distance vector set is transformed to be lexicographically positive, which we know is possible from the proof above for distance vectors. We then construct U' by skewing U, which does not change the lexicographic positiveness of distance vectors. The first row of U' is U. The second row of U' is the second row of U skewed with respect to the first row of U' so that both elements in any given column or both either nonnegative or nonpositive. This is always possible by choosing a sufficiently large skewing factor. We repeat this process for successive rows, so that the matrix U' has all elements in any given column either all nonpositive or all nonnegative.

We now show that $U'\vec{d} \succ \vec{0}$ for all $\vec{d}$. Consider the first component of $U'\vec{d}$:

$$u'_{1,1}d_1 + \cdots + u'_{1,n}d_n$$

Using component arithmetic, the lower bound of this expression is

$$u'_{1,1}x^1_1 + \cdots + u'_{1,n}x^1_n$$

where

$$x^1_k = \begin{cases} d^{\min}_k & \text{if } u'_{1,k} > 0 \\ d^{\max}_k & \text{if } u'_{1,k} < 0 \\ \text{any integer in } [d^{\min}_k, d^{\max}_k] & \text{if } u'_{1,k} = 0. \end{cases}$$

Note that if $u'_{1,k} > 0$ then $d^{\min}_k \neq -\infty$, because otherwise some $\vec{e} \in \mathcal{E}(\vec{d})$ can be chosen such that $U'\vec{e}$ has an arbitrarily negative first component. Likewise, if $u'_{1,k} < 0$, then $d^{\max}_k \neq \infty$. Thus, all the x^1_k are integral and $\vec{x}^1 = (x^1_1, \cdots, x^1_n) \in \mathcal{E}(\vec{d})$. This means that $U'\vec{x}^1$ has a nonnegative first component, and so the first component of $U'\vec{d}$ has a nonnegative minimum. If the first component of $U'\vec{d}$ is positive, then this dependence is satisfied and the proof is complete. So we assume that the first component has a minimum of zero.

Consider the second component of $U'\vec{d}$:

$$u'_{2,1}d_1 + \cdots + u'_{2,n}d_n$$

Using component arithmetic, the lower bound of this expression is

$$u'_{2,1}x^2_1 + \cdots + u'_{2,n}x^2_n$$

where

$$x^2_k = \begin{cases} d^{\min}_k & \text{if } u'_{2,k} > 0 \\ d^{\max}_k & \text{if } u'_{2,k} < 0 \\ x^1_k & \text{if } u'_{2,k} = 0. \end{cases}$$

We first show that if $u'_{2,k} > 0$, then $d^{\min}_k \neq -\infty$. If $u'_{2,k} > 0$, then the k column of U' is nonnegative, and so either $u'_{1,k} > 0$ or $u'_{1,k} = 0$. If $u'_{1,k} > 0$, then $d^{\min}_k \neq -\infty$ as discussed above. If $u'_{1,k} = 0$, then not only is the first component of $U'\vec{x}^1$ zero, as discussed above, but the first component of that product is equal to zero for all x^1_k. If $d^{\min}_k = -\infty$, then x^1_k could be arbitrarily negative, forcing the second component of $U'\vec{x}^1$ to be negative, and thus forcing $U'\vec{x}^1$ to be lexicographically negative. Since $\vec{x}^1 \in \mathcal{E}(\vec{e})$, and therefore $U'\vec{x}^1 \succ \vec{0}$, this is a contradiction. Thus, $d^{\min}_k \neq -\infty$. Likewise, if $u'_{2,k} < 0$, then $d^{\max}_k \neq \infty$.

Thus, all the x^2_k are integral (noninfinite), and the distance vector $\vec{x}^2 = (x^2_1, \ldots, x^2_n)$ is in $\mathcal{E}(\vec{d})$. Also, the first component of $U'\vec{x}^2$ is zero, since $\vec{x}^2$ also meets the requirements to be $\vec{x}^1$. Since $U'\vec{x}^2 \succ \vec{0}$ and has a zero first component, this means that $U'\vec{x}^2$ has a nonnegative second component. But, the second component of $U'\vec{x}^2$ is exactly $u'_{2,1}x^2_1 + \cdots + u'_{2,n}x^2_n$, the lower bound of $U'\vec{d}$, so the second component of $U'\vec{d}$ has a nonnegative minimum. If the second component of $U'\vec{d}$ is positive, then this dependence is satisfied and the proof is complete. If it is zero, then we continue this process, showing that the first nonzero minimum of $U'\vec{d}$ must be strictly positive.

Theorem B.4: Suppose there exists a unimodular transformation T such that $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$. Let S be spanned by $\{\vec{s}_1, \cdots, \vec{s}_{|S|}\}$, where $\forall \vec{d} \in D : \vec{s}_i \cdot \vec{d} = 0$ for $i = 1, \cdots, |S|$. Then there exists a unimodular transformation U such that $\forall \vec{d} \in D : U\vec{d} \succ \vec{0}$ and the first $|S|$ rows of U span S.

Proof: Apply Theorem B.3 to construct a legal transformation U_1 from T with first row $\vec{s}_1$. Further application of Theorem B.3 to construct a legal transformation U_j from U_{j-1} with first row $\vec{s}_j$ results in the first j rows of U_j of the form

$$\begin{bmatrix} \vec{s}_j \\ k_{1,0}\vec{s}_j + k_{1,1}\vec{s}_{j-1} \\ k_{2,0}\vec{s}_j + k_{2,1}\vec{s}_{j-1} + k_{2,2}\vec{s}_{j-2} \\ \vdots \end{bmatrix}.$$

The matrix $U_{|S|}$ meets the requirements for U in the theorem. □

C. Uniqueness of Outermost Fully Permutable Loop Nest

Theorem B.5: Suppose T and W are legal unimodular transformations, given dependence vectors $\vec{d} \in D$. Then there exists a legal unimodular transformation U such that the outermost fully permutable nest of U contains rows spaning all rows in the outermost fully permutable nests of both T and W.

Proof: Let row r of T and W be $\vec{t}_r$ and $\vec{w}_r$, respectively, and let the number of rows in the outermost fully permutable nest of T and W be p_T and p_W, respectively. If the first p_W rows of W are all linearly dependent upon the first p_T rows of T, then T meets the conditions for U, and we are done. Otherwise, we choose a row $\vec{w}$ from $\vec{w}_1$ through $\vec{w}_{p_W}$ that is linearly independent of the first p_T rows of T. We construct a unimodular matrix U_1 that is legal and has its first $p_T + 1$ rows both fully permutable and spanning $\vec{w}$ and $\vec{t}_1$ through $\vec{t}_{P_T}$. By repeating this process, we place additional linearly independent rows from the outermost fully permutable nest of W until we have constructed the desired matrix U.

We apply the construction used in the proof of Theorem B.3 to make $\vec{w}$ the first row of a new legal transformation U_1. As a byproduct of this construction, the first $p_T + 1$ rows of U_1 are of the form

$$\begin{bmatrix} \vec{w} \\ k_{1,0}\vec{w} + k_{1,1}\vec{t}_1 \\ k_{2,0}\vec{w} + k_{2,1}\vec{t}_1 + k_{2,2}\vec{t}_2 \\ \vdots \end{bmatrix}$$

where $k_{j,j} > 0$ for all $j = 2, \ldots, p_T + 1$. The first row of U_1 meets the condition for full permutability, because $\vec{w} \cdot \vec{d} \geq 0$. If $k_{1,0} \geq 0$, then the second row of U_1 will also be in the same fully permutable nest as the first row, since $(k_{1,0}\vec{w} + k_{1,1}\vec{t}_1) \cdot \vec{d} \geq 0$. If $k_{1,0} < 0$, then we only need skew the second row of U_1 with respect to the first row by a factor of $\lceil -k_{1,0} \rceil$ to add the second loop into the outermost fully permutable nest. To include rows 3 through $p_T + 1$ of U_1 in the outermost fully permutable nest, we repeat the same pattern. □

D. Producing Maximal Degree of Parallelism

Theorem B.6: An algorithm that finds the maximum coarse grain parallelism, and then recursively calls itself on the inner loops, produces the maximum degree of parallelism possible.

Proof: The proof is by induction on the number of fully permutable nests. The base case is when the entire loop nest is one fully permutable nest. Our algorithm finds $n - 1$ degrees of parallelism in an n-dimensional loop, unless they are all DOALL loops, in which case it finds n degrees of parallelism. The algorithm produces the maximum degree of parallelism possible by Theorem B.2

Suppose our algorithm finds $k > 1$ fully permutable nests. Let there be n total loops in the nest. We use F to denote the set of loops in the outermost fully permutable loop nest, and F' to denote the set of remaining loops. Let f be the number of loops in set F. The f outermost loops could not all be DOALL loops because otherwise the two outermost loop nests could be merged. Thus the outermost nest has $f - 1$ degrees of parallelism. Suppose the algorithm finds f' degrees parallelism in F', and by the induction hypothesis, f' is the maximum degree of parallelism available given the outer loops F. Thus, the total degree of parallelism is $f + f' - 1$.

Let $(I_1, \cdots, I_n)$ be the loop nest transformed to contain the maximum number of DOALL loops. Let I_m be the first sequential loop within the nest. Then loops $I_1, \cdots, I_{m-1}$ are DOALL loops, and so loops I_1 to I_m are fully permutable and must be spanned by F, by Theorem B.5. Therefore, $m \leq f$, and the first m loops contain $m-1$ degrees of parallelism. The remaining loops include $f - m$ loops from F, and the loops in F'. Since the dependences to be satisfied are a superset of those considered when parallelizing only F', the maximum degree of parallelism of the remaining loops cannot exceed $(f - m) + f'$. Thus, the total degree of parallelism cannot exceed $f + f' - 1$. □

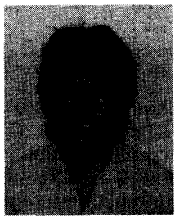
ACKNOWLEDGMENT

We would like to thank the other members of the SUIF compiler group at Stanford University for providing a context in which this research could be carried out. The team includes S. Amarasinghe, J. Anderson, J. Hennessy, D. Maydan, K. Pieper, M. D. Smith, and S. Tjiang.

REFERENCES

[1] R. Allen and K. Kennedy, "Automatic translation of FORTRAN programs to vector form," *ACM Trans. Programming Languages Syst.*, vol. 9, no. 4, pp. 491–542, 1987.
[2] U. Banerjee, "Data dependence in ordinary programs," Tech. Rep. 76–837, Univ. of Illinois Urbana–Champaign, Nov. 1976.
[3] ——, *Dependence Analysis for Supercomputing.* Boston, MA: Kluwer Academic, 1988.
[4] ——, "A theory of loop permutations," in *Proc. 2nd Workshop Languages Compilers Parallel Computing*, Aug. 1989.
[5] ——, "Unimodular transformations of double loops," in *Proc. 3rd Workshop Languages Compilers Parallel Computing*, Aug. 1989.
[6] R. Cytron, "Compile-time scheduling and optimization for asynchronous machines," Ph.D. dissertation, Univ. of Illinois at Urbana–Champaign, 1984.
[7] J.-M Delosme and I. C. F. Ipsen, "Efficient systolic arrays for the solution of Toeplitz systems: An illustration of a methodology for the construction of systolic architectures in VLSI," Tech. Rep. 370, Yale Univ. 1985.
[8] J. A. B. Fortes and D. I. Moldovan, "Parallelism detections and transformation techniques useful for VLSI algorithms," *J. Parallel Distributed Comput.*, vol. 2, pp. 277–301, 1985.
[9] K. Gallivan, W. Jalby, U. Meier, and A. Sameh, "The impact of hierarchical memory systems on linear algebra algorithm design," Tech. Rep., Univ. of Illinois, 1987.
[10] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," *J. Parallel and Distributed Comput.*, vol. 5, pp. 587–616, 1988.

[11] F. Irigoin, "Partitionnement des boucles imbeiquees: Une technique d'optimisation pour les programmes scientifiques," Ph.D. dissertation, Université Paris-VI, June 1987.
[12] F. Irigoin and R. Triolet, "Computing dependence direction vectors and dependence cones," Tech. Rep. E94, Centre D'Automatique et Informatique, 1988.
[13] ——, "Supernode partitioning," in *Proc. 15th Annu. ACM SIGACT–SIGPLAN Symp. Principles Programming Languages*, Jan. 1988, pp. 319–329.
[14] ——, "Dependence approximation and global parallel code generation for nested loops," in *Parallel Distributed Algorithms*, 1989.
[15] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. ACM SIGPLAN '88 Conf. Programming Language Design Implementation*, June 1988, pp. 318–328.
[16] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *Proc. ACM SIGPLAN '91 Conf. Programming Language Design Implementation*, June 1991, pp. 1–14
[17] A. Porterfield, "Software methods for improvement of cache performance on supercomputer applications," Ph.D. dissertation, Rice Univ., May 1989.
[18] P. Quinton, "The systematic design of systolic arrays," Tech. Rep. 193, Centre National de la Recherche Scientifique, 1983.
[19] ——, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *Proc. 11th Annu. Int. Symp. Comput. Architecture*, June 1984.
[20] H. B. Ribas, "Automatic generation of systolic programs from nested loops," Ph.D. dissertation, Carnegie Mellon Univ., June 1990.
[21] R. Schreiber and J. Dongarra, "Automatic blocking of nested loops," 1990.
[22] C.-W. Tseng and M. J. Wolfe, "The power test for data dependence," Tech. Rep., Rice COMP TR90–145, Rice Univ., Dec. 1990.
[23] M. E. Wolf, "Improving parallelism and data locality in nested loops," Ph.D. dissertation, Stanford Univ., 1991, in preparation.
[24] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proc. ACM SIGPLAN '91 Conf. Programing Language Design Implementation*, June 1991, pp. 30–44.
[25] M. J. Wolfe, "More iteration space tiling," in *Proc. Supercomputing '89*, Nov. 1989.
[26] ——, *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: MIT Press. 1989.



Michael E. Wolf received the B.A. degree in physics and applied mathematics from the University of California, Berkeley, in 1984, and the M.S. degree in computer science from Stanford University, Stanford, CA, in 1987.

He is currently a Ph.D. degree candidate in the Department of Computer Science, Stanford University. His research interests include compiler techniques to improve locality and parallelism.

Monica S. Lam (S'84–M'87) received the B.S. degree from the University of British Columbia, in 1980, and the Ph.D. degree in computer science from Carnegie Mellon University, in 1987.

She has been an Assistant Professor in the Department of Computer Science, Stanford University, since 1988. Her reserch interests are in parallel computer systems, including issues in architecture, compilers, and languages. She was one of the chief architects and compiler designers for the CMU Warp machine and the CMU–Intel iWarp. She is currently leading Stanford's parallel compiler research project.