This is to certify that the

dissertation entitled

MATRIX-BASED REPRESENTATIONS OF LOOP TRANSFORMATIONS

presented by

David Raymond Chesney

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Major professor

Date 2/8/95

MATRIX-BASED REPRESENTATIONS OF LOOP TRANSFORMATIONS

By

David Raymond Chesney

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1995

ABSTRACT

MATRIX-BASED REPRESENTATIONS OF LOOP TRANSFORMATIONS

By

David Raymond Chesney

The execution speed of source code on a parallel architecture machine is bound by many factors, including the ability of a parallel compiler to determine parallelism in the target source code. Most of the available parallelism in a source code is contained in the loops and is exploited by applying a sequence of loop transformations. Different methods of representing and ordering sequences of transformations have been developed, including the use of unimodular transformations. A significant feature of the unimodular approach is that it *unifies* a kernel set of seemingly independent loop transformations into the familiar linear algebra domain. The unimodular approach has been applied to *loop permutation*, *loop reversal*, and *loop skewing* in perfectly nested loops.

This research extends matrix-based representations of loop transformations to include a wider family of loop transformations applied to a broad range of source code structures. First, a matrix-based representation of additional loop transformations has been developed, namely *loop normalization*, *loop blocking*, *strip mining*, *cycle shrinking*, *loop collapsing*, *loop coalescing*, *loop fission*, and *loop fusion*. Second, the application of the extended and kernel transformations is generalized to handle both perfectly and imperfectly nested loops. Finally, properties of the original unimodular transformations, including composition, one-time mapping, and the evaluation of parallelism are preserved in the extended model.

To S.T., ILYTTMABA

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Betty H.C. Cheng, for her assistance during my research. Early in my academic career she told me that earning a Ph.D. is like a roller-coaster ride. Thank you, Dr. Cheng, for believing in me in both the highs and the lows of my Ph.D. program.

Thank you also to my committee. Dr. Lionel Ni once said that the key to success in a Ph.D. is finding the question. After that, the answer comes easy. His assistance in helping me find both the question and the answer were invaluable. Dr. Diane Rover helped me both technically, and by showing me that it is possible to balance family and academics. She is a wonderful role model. Dr. David Yen ably assisted on my committee by adding a mathematical perspective.

The pursuit of my Ph.D. would not have been possible without the General Motors Corporation Fellowship. Mr. Dennis Meyers served in the non-always-easy position of corporate mentor during the fellowship. Our numerous early morning meetings led to sound career advice and opportunities. Thank you, Dennis.

Several other faculty and staff stand out. Dr. Carl Page sowed the seeds of my initial interest in computer science, and saw to it that I had a solid foundation. Dr. Richard Enbody is a true friend and has given me direction in both academic and non-academic matters. I have a history with Mrs. Lora Mae Higbee from before my time in the Computer Science Department. She has been a wonderful "aunt" and listener throughout. Mrs. Linda Moore has always exuded genuine warmth and able assistance on my frequent visits to her office.

Thank you to my friends and fellow students. I feel fortunate to be a part of such a remarkable group. I will truly miss our lunch and office conversations dealing with such topics as SPMD machine operation, LaTeX errors, and platypus breasts. By name, they are Andy Fung, Edgar Kalns, Barb Birchler, Gerry Gannod, Steve Turner, Marie-Pierre Dubuisson, Joe Sharnowski, Christian Trefftz, Tim Newman, Jon Engelsman, Jay Kusler, Steve Schafer, Enoch Wang and Oliver Christ. Special thanks to Maureen Galsterer and Linda Cherry for sharing the commute and providing rich conversation. Ron Sass offered critical review of my work that strengthened my overall approach and the potential value of my research. Thanks, Ron.

My parents, Dale A. and Beverly E. Chesney taught me how to dream big dreams. Their love and support has always been an inspiration to me. It's funny! Even as an adult, I still need to make them proud.

My daughter, Mairin, came along midway through my academic career. Her giggles and hugs added more to any success at my research than any amount of time at a workstation. Mairin, I love you dearly, and being your (and your siblings, wink! wink!) Daddy is my true vocation.

Finally, and most importantly, thank you to my wife, Jean. I don't know if you realized what the journey would be like when you married a dreamer. You are the absolute center of my life and there is no possible way that I could have completed my Ph.D. without you. I love you and thank you, from the bottom of my heart.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

The speed of executing source code on a parallel computer is *theoretically* limited by the number of processors [1, 2]. However, a *practical* limitation is the ability of a compiler for a parallel machine to recognize and exploit parallelism and data locality. In order to parallelize and localize data, a compiler is often more complex for parallel machines than for sequential machines. Methods need to be developed that easily and consistently exploit parallelism and locality, which leads directly to faster execution on parallel architecture machines.

Most of the available parallelism in target source code is in the loops and certain combinations of transformations to the loops maximize parallelism or data locality goals. Examples of the loop transformations include *loop permutation*, *loop reversal*, *loop skewing*, *loop fission*, *loop fusion*, *loop blocking* (*tiling*), *strip mining*, *cycle shrinking*, *loop collapsing*, *loop coalescing*, and *loop normalizing*. Individually, the transformations are fairly well understood, however, they are often treated independently.

Thesis Statement.

The purpose of this research is to extend matrix-based representations of loop transformations to include loop normalization, loop blocking, tiling, cycle shrinking, strip mining, loop collapsing, loop coalescing, loop fission, and loop fusion. The transformations are applied to both imperfectly and perfectly nested loops, and preserve attractive properties of the tradi-

tional matrix-based models. New algorithms are developed that use the kernel and extended transformation set to obtain both machine-independent and machine-dependent goals.

1.1 Motivation

A *unified* framework that represents a broad spectrum of transformation techniques is essential to the usefulness of a parallel compiler. An integrated framework facilitates the user's ability to manipulate and transform source code within a common context. Recently, several generalized transformation schemes have been proposed to unify the seemingly independent transformations, including *unimodular matrices* [3, 4], *non-singular matrices* [5, 6], *schedules* [7, 8], and other general approaches [9, 10, 11].

Each generalized model has its own set of strengths and weaknesses. One weakness of the generalized models is that only a limited number of loop transformations are represented. As an example, unimodular transformations represent loop permutation, reversal, and skewing. Second, there is typically an explicit limitation on the source code structure before the generalized model is applied. That is, the models typically require perfect loop nesting. Third, mapping rules for the dependences are often imprecise. Accurate dependence information is paramount to the evaluation of any transformation sequence. The accuracy needs to be maintained throughout the transformation sequence in order for parallelism goals to be evaluated.

A major strength of each transformation model is the unification of a set of loop transformations, which allows a user to manipulate different loop transformations within a common context. Unimodular transformations offer three additional strengths: one-time mapping of initial source code into transformed source code; composability; and the ability to evaluate parallelism goals using an abstract representation of the source code and transformation sequence.

Unimodular transformations are a set of loop transformations that are represented by matrices with specific properties. Specifically, loop permutation, reversal, and skewing are all represented by unimodular matrices. A sequence of transformations is represented by the

product of the individual transformations that comprise it, and is called the transformation matrix T.

The user can determine if the transformed source code is legal based upon properties of the product of the transformation matrix T and a vector representation of the dependences D is the source code. Source code parallelism is also determined from the product of the transformation matrix T and the dependences D. The source code is *mapped* from initial to final form, given the transformation matrix T. That is, there exists an algorithm for source code mapping based upon the matrix representing the sequence of transformations.

1.2 Contributions

The major contributions of this research are: 1) to extend the matrix-based representations of loop transformations; 2) to generalize the kernel and extended set of transformations to apply to both perfect and imperfect loop nests; and 3) to develop new algorithms that are based upon combinations of the transformations to achieve goals relevant to a parallel compiler. The research presented in this dissertation preserves the strengths of unimodular transformations, yet overcomes the weaknesses of other generalized models [12].

First, this dissertation describes an extension to the unimodular approach to include a wider class of loop transformations. Specifically, five new families of transformation techniques are described in terms of a matrix-based framework: loop normalization [13, 14, 15], loop blocking [4, 14, 15, 16, 17, 18, 19], loop collapsing [14, 15, 18], loop fission [17, 18, 19, 20, 21], and loop fusion [14, 15, 17, 18, 19].

Loop normalization is a transformation that transforms source code so that the lower bounds and step sizes of a loop are both one. Normalized loops are often a precondition for kernel transformations and for many extended transformations that are discussed in this dissertation. Loop fission is a loop transformation technique that divides a set of program statements within a nested loop into two separate nested loops. The inverse of loop fission is loop fusion, in which two adjacent nested loops are combined into one [22]. Loop blocking includes tiling, strip mining, and cycle shrinking; and this transformation increases the

depth of a nested loop. Loop collapsing includes loop coalescing and is the inverse of loop blocking because it decreases the depth of a nested loop [23].

Mapping for the source code and dependences is explicitly defined for each extended transformation. The rules for source code mapping show how the initial source code is mapped to its final form. Accurate dependence mapping rules ensure that the transformed source code is semantically equivalent to the initial source code and allows for evaluation of potential parallelism.

Another important advantage of the extended approach is the ability to transform imperfectly nested loops. *Imperfect loop nests* do not have all assignment statements contained in the deepest loop. The generalized unimodular framework presented in this dissertation enables both the kernel set (loop permutation, reversal, and skewing) and the extended set (loop normalization, fission, fusion, blocking, and collapsing) of loop transformations to be applied to both perfect and imperfect loop nests.

Loop transformations are used for numerous purposes in a parallel compiler. For example, loop permutation and skewing may be used to obtain parallelism. Loop blocking may result in improved data locality or reduced barrier synchronization. Combining loop blocking, permutation, and skewing enables parallelism, data locality, and barrier synchronization goals to be attained within a matrix-based framework. That is, extending the unimodular approach to include additional transformations broadens the potential advantages to include those of the extended transformation set.

Several new algorithms are introduced that combine the kernel and extended transformation sets on machine-independent and machine-dependent goals. Machine-independent goals include data locality and parallelism, and machine-dependent goals include transformation of source code for a vector or shared-memory machine. The algorithms combine the loop transformations to obtain goals that are beyond the scope of the kernel transformation set alone.

The current unimodular approach is a theoretically elegant method for applying a limited number of transformations to a restricted set of source code structures. The generalization

presented here broadens the types of transformations that are applied to a wider class of source code structures.

1.3 Organization

The remainder of this dissertation is organized as follows. Chapter 2 gives definitions and background information used throughout the dissertation. Chapter 3 discusses the generalization of the kernel transformations (loop permutation, reversal, and skewing) to include imperfectly nested loops and also loop normalization. The next chapter discusses two additional transformations that modify the nest depth of a nested loop, namely loop blocking and collapsing. Chapter 5 discusses loop fission and fusion, which are the last of the extended transformations. Algorithms that attain machine-independent and machine-dependent goals using the kernel and extended transformation sets are discussed in Chapter 6. Chapter 7 uses the *object modeling technique* to describe the implementation framework of the system. Related work is discussed in Chapter 8. Finally, conclusions and future investigations are discussed in Chapter 9.

Chapters 3 through 5 describe matrix representations of individual loop transformations, and the format of all of the chapters is similar. First, the transformation is briefly described and an example is given. Second, the legality of the transformation is discussed in terms of *invariant, explicit*, and *nesting* preconditions. Next, the effect of the loop transformation on the distance vector set D is discussed. Accurate mapping of D is essential to evaluation of overall legality, parallelism, and other transformation goals. Finally, the source code mapping is discussed.

CHAPTER 2

Background and Definitions

This section gives background and definitions for terms used throughout the dissertation. First, the terminology related to source code structure is discussed. Next, the taxonomy of perfect and imperfect loop nests is described. Third, dependences and their properties are discussed. Next, fundamentals of unimodular transformations are described, followed by the global pre- and postcondition for applying a sequence of matrix-based transformations.

2.1 Source Code Structure

This section discusses the explicit notation used throughout the dissertation to describe loop bounds and subscript expressions.

Loop Bound Notation

In general, a loop nest is in the form shown in Figure 2.1, where n is the loop nest depth. The entire iterative statement "**for** $I_i = lb_i, ub_i, s_i$" is referred to as L_i. I_1 through I_n are *index variables*, and lb_i, ub_i, and s_i are the *lower bound*, *upper bound*, and *step size*, respectively, for the loop with index variable I_i. In order to prevent ambiguity, the subscripts for the upper bound, lower bound, and step size may contain the name of the index variable with which they are associated. As an example, lb_{I_1} is equivalent to lb_1.

If there is an n-nested loop, then the loop bound set B has n loop bound vectors $\vec{b}$. Each $\vec{b}$ has three loop bound expressions: the lower bound, upper bound, and step size. As

L_1 **for** $I_1 = lb_1, ub_1, s_1$
L_2 **for** $I_2 = lb_2, ub_2, s_2$
$\vdots$
L_n **for** $I_n = lb_n, ub_n, s_n$
S_1 ARR$[sub_1, sub_2, ..., sub_m] = \ldots$

Figure 2.1. Sample code: general form of a nested loop.

an example, the source code given in Figure 2.1 has the loop bound set:

$$B = \{(lb_1, ub_1, s_1), (lb_2, ub_2, s_2), \ldots, (lb_n, ub_n, s_n)\}.$$

The first vector in B is $\vec{b_1} = (lb_1, ub_1, s_1)$ and refers to the outermost loop. The last vector in B is $\vec{b_n} = (lb_n, ub_n, s_n)$ and refers to the innermost loop.

A restriction on the form of the lower bounds, upper bounds, and step sizes in the loop nest is that each lb_i, and ub_i, $1 \leq i \leq n$, is a linear function of the surrounding index variables I_j, such that $j < i$; and that s_i is an integer constant. Then each lb_i is in the form:

$$a_{i,1}I_1 + a_{i,2}I_2 + \ldots + a_{i,j}I_j + c_i,$$

where $j < i$, $a_{i,j}$ is the coefficient of index variable I_i with respect to I_j, and c_i is an integer constant. Similarly, each ub_i is in the form:

$$b_{i,1}I_1 + b_{i,2}I_2 + \ldots + b_{i,j}I_j + d_i,$$

where $j < i$, $b_{i,j}$ is the coefficient of index variable I_i with respect to I_j, and d_i is an integer constant. As an example, suppose that we have a loop in the form "**for** $I_3 = 2I_1+6, 8I_2, m$", then:

$$a_{3,1} = 2; \quad b_{3,1} = 0; \quad s_2 = m;$$
$$a_{3,2} = 0; \quad b_{3,2} = 8;$$
$$c_2 = 6; \quad d_2 = 0.$$

Subscript Expression Notation

In a loop nest, S_i refers to an entire assignment statement. If **ARR** is the name of an m-dimensional array, then each sub_i, $1 \leq i \leq m$, is referred to as either an *array subscript*, *subscript expression*, or *reference to index variables*. Note that m does not necessarily equal n. That is, the loop nest depth and dimension of the array are not necessarily the same.

An intuitive representation of an array **ARR** is using *array index function form* [24]. An m-dimensional array **ARR** in a n-nested loop is in the form $\mathbf{ARR}[sub_1, sub_2, \ldots, sub_m]$, where each sub_i is a linear function of the loop index variables of the loops plus a constant term. The subscript expressions in array **ARR** are represented in the form:

$$\begin{bmatrix} H_{1,1} & \cdots & H_{1,n} \\ H_{2,1} & \cdots & H_{2,n} \\ \vdots & & \vdots \\ H_{m,1} & \cdots & H_{m,n} \end{bmatrix} \times \begin{bmatrix} I_1 \\ \vdots \\ I_n \end{bmatrix} + \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

where the $m \times n$ matrix H is called the *array index function*. In general, the representation of an array in terms of an array index function is in the form $[H][I_i] + [c_j]$, and is called *array index function form*. As an example, the assignment statement S_1 shown in Figure 2.2 has the array index function form shown to the right of the source code.

L_1 **for** $I_1 = 1, ub_1$
L_2 **for** $I_2 = 1, ub_2$
S_1 **ARR**$[3I_1, I_1 + 2I_2 + 3, 4] = \ldots$

$$\begin{bmatrix} 3 & 0 \\ 1 & 2 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 3 \\ 4 \end{bmatrix}$$

Figure 2.2. Sample code and array index function for statement S_1.

Two array references have *uniformly generated references* if they refer to the same array and have the same array index function H. A *uniformly generated set* is an equivalence class, such that any two members of the set have uniformly generated references. There is little potential for reuse in an array that has non-uniformly generated references [24]. That

is, arrays with the same name, that also have the same array index function H, will likely have data locality that can be exploited.

If all $H_{i,j}$ and c_i for a subscript expression sub_i are integer constants, then sub_i is *linear*. Otherwise, sub_i is *nonlinear*. If all subscript expressions in an m-dimensional array **ARR** are linear, then **ARR** is linear. If some subscript expressions are nonlinear, then **ARR** is partially linear. If no subscript expressions are linear, then **ARR** is nonlinear. Finally, if index variable I_i, $1 \leq i \leq n$, appears in more than one sub_j, $1 \leq j \leq m$, then **ARR** has *coupled subscripts*, where n is the loop nest depth and m is the dimension of the array [25].

2.2 Perfect and Imperfect Loop Nests

A *loop nest hierarchy* is a representation of the structure of a nested loop using a rooted tree, where the parent-child relation is defined in terms of loop nesting and the root is the outermost loop. Specifically, loop L_i is the parent of loop L_j if loop L_i is the next outer level of loops that completely encloses loop L_j. For example, in Figure 2.3(a), L_1 is the parent of L_2 and L_2 is the parent of L_3. In Figure 2.3(b), L_4 is the parent of L_5 and L_5 is the parent of L_6. Finally, in Figure 2.3(c), L_7 is the parent of both L_8 and L_9. Note that in the source code in Figures 2.3(b) and 2.3(c), there are assignment statements between the respective **for**'s and the **endfor**'s. (The **endfor**'s are included in Figures 2.3(a) through 2.3(c) for clarity. In other source code samples in this dissertation, the location of the **endfor**'s is implied.) Two loops may have the same parent, as shown in Figure 2.3(c); however, no loop may have more than one parent.

Two loops are *perfectly nested* if there are no statements between their opening **for** statements, nor between their respective closing **endfor** statements. A *loop nest* is perfectly nested if each parent-child pair in the loop nest hierarchy is perfectly nested. Note that there is a difference between perfect nesting of *two loops* and perfect nesting of an entire *loop nest*. A loop nest that is not perfectly nested is *imperfectly nested*.

For example, L_1 and L_2 are perfectly nested, as are L_2 and L_3 . Therefore, the entire loop nest in Figure 2.3(a) is perfectly nested. L_4 and L_5 are imperfectly nested in Figure

```
L1  for I1 ...
L2   for I2 ...
L3    for I3 ...
S1     var1 = ...
S2     var2 = ...
S3     var3 = ...
      endfor
     endfor
    endfor
```

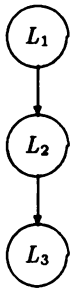

Figure 2.3(a). Sample code and loop nest hierarchy: *perfect* nesting.

```
L4  for J1 ...
S1   var1 = ...
L5   for J2 ...
L6    for J3 ...
S2     var2 = ...
S3     var3 = ...
      endfor
     endfor
    endfor
```

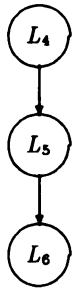

Figure 2.3(b). Sample code and loop nest hierarchy: *imperfect-only-child* nesting.

```
L7  for K1 ...
S1   var1 = ...
L8   for K2 ...
S2    var2 = ...
     endfor
L9   for K3 ...
S3    var3 = ...
     endfor
    endfor
```

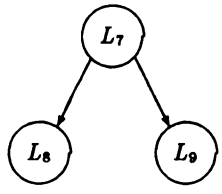

Figure 2.3(c). Sample code and loop nest hierarchy: *imperfect-sibling* nesting.

2.3(b), however, L_5 and L_6 are perfectly nested. Therefore, the loop nest in Figure 2.3(b) is imperfectly nested. Similarly, L_8 and L_9 are imperfectly nested in L_7, and the entire loop nest is imperfectly nested in Figure 2.3(c).

The taxonomy for perfect and imperfect loop nests is derived from the form of the loop nest hierarchy. *Perfect* loop nests (Figure 2.3(a)) are rooted trees of degree one with no statements between opening **for** 's and closing **endfor** 's of any parent-child loops. *Imperfect-only-child* loop nests (Figure 2.3(b)) are rooted trees with degree one with at least one assignment statement between a parent-child pair of **for** 's or between a parent-child pair of **endfor** 's. Thus, statement S_1 in Figure 2.3(b) makes the loop nest imperfect-only-child. Finally, *imperfect-sibling* loop nests (Figure 2.3(c)) are rooted trees with degree greater than one. The relationship between these three different forms of loop nests is shown in Figure 2.4.

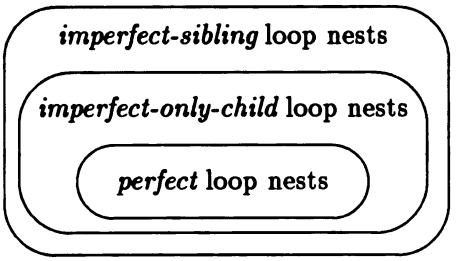

Figure 2.4. Relationship of *perfect* and *imperfect* loop nests.

The applicability of unimodular transformations is dependent upon the distinction between perfect, imperfect-only-child, and imperfect-sibling loop nests. That is, currently unimodular transformations are applied only to perfect loop nests. This research shows how to apply transformation matrices to imperfect-only-child loop nests, and how to transform many imperfect-sibling loop nests into imperfect-only-child loop nests.

2.3 Dependences

In any assignment statement, the variable on the left-hand side (**LHS**) is being defined (*def*), and any variables on the right-hand side (**RHS**) are being used (*use*). The three types of dependences: *flow*, *anti*, and *output*, are defined in terms of the chain of *def*'s and *use*'s that result in a dependence. Flow dependence is defined as a *def-use* chain; anti dependence is defined as a *use-def* chain; and output dependence is defined as a *def-def* chain. Symbols representing flow, anti, and output dependence are δ^f, δ^a, and δ^o, respectively, where δ represents one or more occurrences of these types. That is, if the relation δ^f, δ^a, or δ^o holds, then δ holds.

Source code that illustrates each type of dependence is given in Figure 2.5: S_5 is flow dependent upon S_1 due to A ($S_1\ \delta^f\ S_5$); S_3 is output dependent upon S_2 due to B ($S_2\ \delta^o\ S_3$); S_4 is anti dependent upon S_3 due to G ($S_3\ \delta^a\ S_4$); and S_2 and S_3 are anti dependent upon S_1 due to B ($S_1\ \delta^a\ S_2$, $S_1\ \delta^a\ S_3$). Note that the order of the statements in the notational

description implies the order in the execution of the source code. That is, $S_1\ \delta^f\ S_5$ implies that statement S_1 must finish execution before statement S_5 begins execution.

S_1 $A = A + B$
S_2 $B = C + D$
S_3 $B = G + 1$
S_4 $G = G + 1$
S_5 $H = A + K$

Figure 2.5. Sample code containing scalar *flow, anti,* and *output* dependence.

Dependences can be represented in two forms: distance vectors and direction vectors. In distance vectors, the components are integral, whereas in direction vectors, the components are symbolic ($<, =$, and $>$).

A *distance vector set* D is a set representation of the dependences within a body of source code. Each member of the set is called a *distance vector* $\vec{d}$. Finally, each component of a distance vector d_i is a *distance.* For an n-nested loop, d_1 is the outermost loop, d_n is the innermost loop, and d_i, $1 < i < n$, refers to the respective loops between L_1 and L_n.

For example, the distance vector set for the loop nest in Figure 2.6 is:

$$D = \{(0,1),(1,0),(1,-1),(0,2),(0,3)\},$$

where the first element of D, (0,1) is the distance vector for the output dependence between S_2 and S_1; the second element (1,0) is the distance vector for the anti dependence between S_3 and S_1; the third element of D, (1,-1) is the distance vector for the anti dependence between S_4 and S_1; the fourth element of D, (0,2) is the distance vector for the anti dependence between S_1 and itself; and the fifth element of D, (0,3) is the distance vector for the anti dependence between S_2 and S_1.

A symbolic direction in a dependence vector describes the temporal nature of dependences between statements. The integer distances in a distance vector explicitly define the

```
L1  for I1 = 1, n
L2   for I2 = 1, n
S1     A[I1, I2] = A[I1, I2 + 2] + B[I1, I2] + C[I1, I2]
S2     A[I1, I2 − 1] = var1
S3     B[I1 − 1, I2] = var2
S4     C[I1 − 1, I2 + 1] = var3
```

Figure 2.6. Sample code containing array *flow, anti,* and *output* dependences.

distance in terms of iteration space between uses of a variable, and distance vectors are used in this dissertation to represent dependence information. It should be noted that direction vectors can be derived from distance vectors, but not vice-versa. As an example, the distance vector (2,-3,0) is equivalent to the direction vector $(<, >, =)$.

2.4 Properties of Dependences

This section formalizes definitions related to dependence vectors and dependence vector sets. A distance vector $\vec{d}$ in an n-nested loop is in the form $(d_1, \ldots, d_i, \ldots, d_n)$. A *loop-independent* dependence is one in which all of the distances in the distance vector are zero:

$$(\forall i : 1 \leq i \leq n : d_i = 0).$$

A *loop-carried* dependence is one in which at least one distance is non-zero:

$$(\exists j : 1 \leq j \leq n : d_j \neq 0).$$

In a loop-carried dependence, the dependence is *carried* at the loop depth of the first non-zero distance. That is, if d_j is the first non-zero distance in $\vec{d}$, then the dependence is carried at depth j.

Each dependence relation exists *from* a statement *to* a statement of source code. An intuitive representation of the dependence relations uses a directed graph $G(V, E)$ called a *dependence graph.* The vertices V are the statements in the source code and the edges E

are the dependences between the statements. Figure 2.7 is the dependence graph for the source code in Figure 2.6, where the arc points in the direction of the dependence.

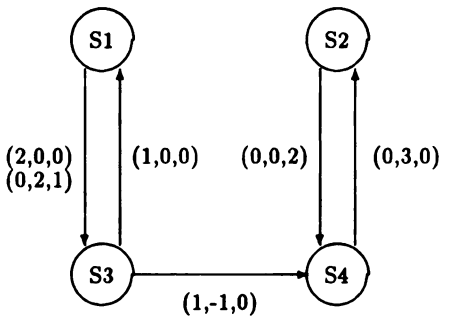

Figure 2.7. Dependence graph (depth = 1).

Dependence graphs are also defined in terms of depths of the carried dependences. A dependence graph at depth i contains all of the dependences that are carried at depth i or greater. The depth 1 dependence graph represents all of the dependences in the source code. As an example, the depth 1 dependence graph for the source code shown in Figure 2.6 is given in Figure 2.7. The depth 2 dependence graph contains all dependences that are carried at a depth of 2 or greater. Thus, the distance vectors {(2,0,0),(1,0,0),(1,-1,0)} are eliminated from the dependence graph in Figure 2.8. Similarly, the depth 3 dependence graph is shown in Figure 2.9.

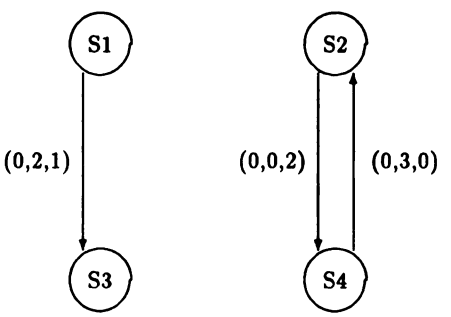

Figure 2.8. Dependence graph (depth = 2).

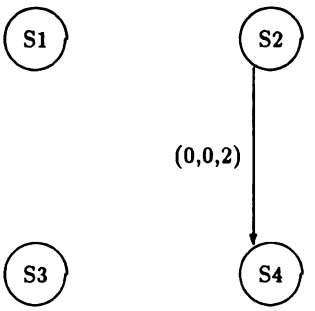

Figure 2.9. Dependence graph (depth = 3).

A *lexicographically positive* distance vector, denoted $\vec{d} \succ \vec{0}$, has some positive distance within the distance vector and all distances preceding the positive distance are nonnegative:

$$\vec{d} \succ \vec{0} :: ((\exists i : 1 \leq i \leq n : d_i > 0) \ \wedge \ (\forall j : 1 \leq j < i : d_j \geq 0)),$$

where $\vec{0}$ represents the zero vector $(0, 0, 0, \ldots, 0)$.

As an example, the distance vector $(0, 0, 2, -1)$ is lexicographically positive because the third distance (d_3) is greater than 0; and the first (d_1) and second (d_2) distances are both greater than or equal to 0. If all of the distance vectors $\vec{d}$ in a distance vector set D are lexicographically positive, then D is lexicographically positive.

Loops L_i through L_j are *fully permutable* if the distance vector set is in the following form:

$$(\forall \vec{d} : \vec{d} \in D : ((d_1, \ldots, d_{i-1}) \succ \vec{0}) \ \vee \ (\forall k : i \leq k \leq j : d_k \geq 0)). \tag{2.1}$$

That is, either the distance vectors are lexicographically positive preceding location i, or are non-negative in the range from d_i to d_j. Fully permutable distance vectors are significant because, as the name implies, any permutation of loops in the range L_i through L_j results in a legal transformation. Suppose that we have a predicate *fully_perm*$(\vec{d}, i, j)$ that evaluates whether distance vector $\vec{d}$ is fully permutable in the range of d_i through d_j. Evaluation of *fully_perm*$(\vec{d}, i, j)$ on $\vec{d}$=(0,1,-3,2), $1 \leq i < j \leq n$ results in the following table:

i	j	*fully_perm*
1	2	yes
1	3	no
1	4	no
2	3	no
2	4	no
3	4	yes

Typically, the objective of a loop transformation sequence is to maximize parallelism, and parallelism is determined based upon properties of the distance vectors $\vec{d}$ in the distance vector set D [4]. Consider a loop nest of depth n, with lexicographically positive distance vectors $(\forall \vec{d} : \vec{d} \in D : \vec{d} \succ \vec{0})$. A loop L_i can be parallelized if either the distance vectors are lexicographically positive preceding location i, or the distance *at* location i is 0. That is, the i^{th} loop is parallelizable if and only if:

$$(\forall \vec{d} : \vec{d} \in D : ((d_1, \ldots, d_{i-1}) \succ \vec{0}) \vee (d_i = 0)).$$

This result is significant because it describes a property for determining available parallelism *based only upon the form of the distance vectors in* D. That is, parallelism is determined from the abstract representation of the source code in terms of the dependence vector set.

The criteria for vectorization relies more heavily on the dependence graph $G(V, E)$ and is less restrictive than the criteria for parallelization. That is, source code that cannot be parallelized may be vectorizable, but not vice-versa. The criteria for vectorization is dependent upon cycles in the dependence graph. Specifically, loop L_i can be vectorized in an assignment statement as long as there are no cycles in the dependence graph carried at depth i.

As an example, loop L_1 of statements S_1, S_2, S_3, and S_4 cannot be vectorized because all of the statements are in cycles in the depth 1 dependence graph shown in Figure 2.7.

Loop L_2 can be vectorized for statements S_1 and S_3 because they are not in a cycle in the depth 2 dependence graph shown in Figure 2.8. However, statements S_2 and S_4 cannot be vectorized. All statements can be vectorized at depth 3, because none are in a dependence cycle as shown in Figure 2.9.

2.5 Unimodular Transformations

The objectives of the unimodular approach are: 1) to represent loop permutation, reversal, and skewing as matrices and 2) to transform initial source code into some final source code that is semantically equivalent to the original source code, but exhibits a desirable property (parallelism, data locality, etc.) [3, 4, 9, 26]. All transformations are represented as operations between a transformation matrix T and the dependence vectors $\vec{d}$ in the dependence vector set D.

A *unimodular* matrix T has the following properties, where **I** is the set of all integers, and $T[i,j]$ refers to the element of T in the i^{th} row and j^{th} column:

1. $n \times n$
2. $(\forall i : 1 \leq i \leq n : (\forall j : 1 \leq j \leq n : T[i,j] \in \mathbf{I}))$
3. $|det\, T| = 1$

This definition states that T is square; all components of T are integers; and the absolute value of the determinant of T is 1.

Three kernel loop transformations represented by unimodular matrices are:

Permutation: interchanging two loops;
Reversal: modifying a loop to increment from the negation of the upper bound to the negation of the lower bound;
Skewing: adding an integer function of an outer loop to the indices of an inner loop.

The matrices that represent permutation, reversal, and skewing are all simple modifications to the identity matrix I. The following are examples of each kernel transformation matrix applied to the distance vector $\vec{d} = (1, 3, -2, 0)$ as shown in the source code in Figure 2.10.

```
L1  for I1 = lb1, ub1
L2   for I2 = lb2, ub2
L3    for I3 = lb3, ub3
L4     for I4 = lb4, ub4
S1      A[I1, I2, I3, I4] = ...
S2      A[I1 − 1, I2 − 3, I3 + 2, I4] = ...

/**D = {(1, 3, −2, 0)}**/
```

Figure 2.10. Sample code: before unimodular transformation.

The 4×4 matrix t_p shown in Figure 2.11(a) is used to represent loop permutation of loops L_1 and L_2. Note that the original lower and upper bounds of loops L_1 and L_2 (Figure 2.10) are swapped by loop interchange, as shown in Figure 2.11(b). Also, the distance vector set after loop permutation of loops L_1 and L_2 is $t_p \times \vec{d} = (3, 1, -2, 0)$.

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

Figure 2.11(a). Unimodular matrix to permute loops L_1 and L_2.

```
L1  for J1 = lb2, ub2
L2   for J2 = lb1, ub1
L3    for J3 = lb3, ub3
L4     for J4 = lb4, ub4
S1      A[J1, J2, J3, J4] = ...
S2      A[J1 − 1, J2 − 3, J3 + 2, J4] = ...

/**D = {(3, 1, −2, 0)}**/
```

Figure 2.11(b). Sample code: after loop permutation of loops L_1 and L_2.

In order to *reverse* loop L_3, the transformation matrix t_r is used as shown in Figure 2.12(a), where $t_r \times \vec{d} = (1, 3, 2, 0)$. Note the differences between the loop bound expressions and array subscripts for index variable J_3 in Figures 2.10 and 2.12(b).

Finally, the matrix shown in Figure 2.13(a) represents *skewing* loop L_2 with respect to loop L_1 by a factor of 2. The dependence vector is multiplied by t_s, where $t_s \times \vec{d} = (1, 5, -2, 0)$.

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

Figure 2.12(a). Unimodular matrix to reverse loop L_3.

L_1 **for** $J_1 = lb_1, ub_1$
L_2 **for** $J_2 = lb_2, ub_2$
L_3 **for** $J_3 = -ub_3, -lb_3$
L_4 **for** $J_4 = lb_4, ub_4$
S_1 $\mathbf{A}[J_1, J_2, -J_3, J_4] = \ldots$
S_2 $\mathbf{A}[J_1 - 1, J_2 - 3, -J_3 + 2, J_4] = \ldots$

$/**D = \{(1, 3, 2, 0)\}**/$

Figure 2.12(b). Sample code: after loop reversal of loop L_3.

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

Figure 2.13(a). Unimodular matrix to skew loop L_2 by $2L_1$.

L_1 **for** $J_1 = lb_1, ub_1$
L_2 **for** $J_2 = lb_2 + 2J_1, ub_2 + 2J_1$
L_3 **for** $J_3 = lb_3, ub_3$
L_4 **for** $J_4 = lb_4, ub_4$
S_1 $\mathbf{A}[J_1, J_2 - 2J_1, J_3, J_4] = \ldots$
S_2 $\mathbf{A}[J_1 - 1, J_2 - 3 - 2J_1, J_3 + 2, J_4] = \ldots$

$/**D = \{(1, 5, -2, 0)\}**/$

Figure 2.13(b). Sample code: after loop skewing of loop L_2 by $2L_1$.

In this dissertation, a superscript is sometimes used on the distance vector set D and the transformation matrix T for clarity purposes. A superscript on the distance vector set D has the following interpretation:

- $D^{(0)}$: the initial distance vector set for the source code before the transformation;
- $D^{(x-1)}$: the transformed distance vector set *before* the x^{th} transformation;
- $D^{(x)}$: the transformed distance vector set *after* the x^{th} transformation;
- $D^{(\Omega)}$: the final, transformed distance vector set.

Similarly, superscripts on the transformation matrix T have the following interpretation:

- $T^{(0)}$: the identity matrix;
- $T^{(x-1)}$: the transformation matrix *before* the x^{th} transformation;
- $T^{(x)}$: the transformation matrix *after* the x^{th} transformation;
- $T^{(\Omega)}$: the final transformation matrix;
- $t^{(x)}$: the elementary transformation matrix representing the x^{th} elementary operation.

The relation between $D^{(i)}$ and $T^{(j)}$ is as follows:

- $T^{(x)} = t^{(x)} \times T^{(x-1)}$: the matrix representing the first x transformations is the product of the x^{th} elementary transformation matrix and the matrix representing the first $x-1$ transformations;
- $D^{(x)} = T^{(x)} \times D^{(0)}$: the transformed distance vector set after x transformations is the product of the transformation matrix representing the first x transformations and the initial distance vector set;
- $D^{(x)} = t^{(x)} \times D^{(x-1)}$: the transformed distance vector set after x transformations is the product of the x^{th} elementary transformation matrix and the distance vector set after $x-1$ transformations.

Unimodular matrices have a *composition* property; that is, sequences of unimodular transformations are also unimodular. As an example, a sequence of loop reversal, permutation, and skewing using the matrices shown in Figures 2.11(a), 2.12(a), and 2.13(a) may be composed into one transformation matrix $T^{(3)} = t^{(3)} \times t^{(2)} \times t^{(1)}$:

$$T^{(3)} = \begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The distance vector set can be transformed in one of two ways: after each kernel transformation ($D^{(x)} = t^{(x)} \times \vec{d}^{(x-1)} \in D^{(x-1)}$); or after the final $T^{(x)}$ is found that represents the entire transformation sequence ($D^{(x)} = T^{(x)} \times \vec{d}^{(0)} \in D^{(0)}$). Continuing with the earlier example, $D^{(3)} = T^{(3)} \times \vec{d}^{(0)} \in D^{(0)}$:

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 3 \\ -2 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ 2 \\ 0 \end{bmatrix}$$

The composition property is significant because a sequence of transformations is represented by a single matrix. Therefore, mapping the distance vector set D and source code need occur only once.

2.6 Global Pre- and Postconditions

Initially, all dependence vectors $\vec{d}^{(0)}$ in the dependence vector set $D^{(0)}$ are lexicographically positive. Formally, this property is described as $(\forall \vec{d}^{(0)} : \vec{d}^{(0)} \in D^{(0)} : \vec{d}^{(0)} \succ \vec{0})$. This condition is referred to as the *global precondition*, because it is true for all distance vectors $\vec{d}^{(0)}$ before any transformations are applied.

The global postcondition must be true *after* all unimodular transformations have been applied, and asserts that all *transformed* dependence vectors $(\vec{d}^{(\Omega)} \in D^{(\Omega)})$ must be lexicographically positive. This condition is formally described as $(\forall \vec{d}^{(\Omega)} : \vec{d}^{(\Omega)} \in D^{(\Omega)} : \vec{d}^{(\Omega)} \succ \vec{0})$, where $D^{(\Omega)} = T^{(\Omega)} \times \vec{d}^{(0)} \in D^{(0)}$. Requiring lexicographically positive distance vectors *after* loop transformation ensures that the source code after transformation is semantically equivalent to the code before transformation.

The global pre- and postconditions are significant because they govern the legality of a sequence of unimodular transformations. That is, all $\vec{d}^{(0)} \in D^{(0)}$ are initially lexicographically positive. Any sequence of transformations, represented by $T^{(\Omega)}$, can be legally applied to the source code, as long as all transformed distance vectors $(T^{(\Omega)} \times \vec{d}^{(0)} \in D^{(0)})$ are lexicographically positive.

2.7 Transforming Source Code

Wolf and Lam [4] describe an approach for mapping source code from initial to final form, given a transformation matrix T. The approach is divided into two portions: mapping assignment statements and mapping loop bounds. Mapping loop bounds is further subdivided into a four-step algorithm. Both assignment statement mapping and loop bound mapping are described below.

Mapping assignment statements requires the inverse of the transformation matrix (T^{-1}), which is used to determine the appropriate linear combination of transformed loop indices to replace the initial loop indices. In matrix form, the following multiplication is used to determine new assignment statements, where I_i are initial loop indices and J_j are new loop

indices:

$$\begin{bmatrix} I_1 \\ \vdots \\ I_n \end{bmatrix} = T^{-1} \times \begin{bmatrix} J_1 \\ \vdots \\ J_n \end{bmatrix}$$

It is also possible to determine new loop indices in terms of the old loop indices in matrix form:

$$T \times \begin{bmatrix} I_1 \\ \vdots \\ I_n \end{bmatrix} = \begin{bmatrix} J_1 \\ \vdots \\ J_n \end{bmatrix}$$

and then solve for each I_i using algebraic substitution. As described later, T representing the extended transformations may be non-square, and T^{-1} is undefined for non-square matrices. Therefore, algebraic substitution, rather than direct matrix-vector multiplication, is used to solve for I_i in terms of J_j. As an example, the assignment statements from the source code shown in Figure 2.10 are mapped as shown in Figure 2.14 using the transformation matrix T above. The boxed expressions represent the algebraic solution of the initial index variables I_i in terms of the transformed index variables J_j. The final step is mapping the transformed assignment statements from the initial assignment statements.

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \end{bmatrix} = \begin{bmatrix} J_1 \\ J_2 \\ J_3 \\ J_4 \end{bmatrix}$$

$J_1 = 2I_1 + I_2$ $\quad \boxed{I_1 = J_2}$ $\quad J_3 = -I_3$ $\quad \boxed{I_4 = J_4}$

$I_2 = J_1 - 2I_1$ $\quad \boxed{I_3 = -J_3}$

$\boxed{I_2 = J_1 - 2J_2}$

$A[I_1, I_2, I_3, I_4] \Rightarrow \quad A[J_2, J_1 - 2J_2, -J_3, J_4]$

$A[I_1 - 1, I_2 - 3, I_3 + 2, I_4] \Rightarrow \quad A[J_2 - 1, J_1 - 2J_2 - 3, -J_3 + 2, J_4]$

Figure 2.14. Transformation of assignment statements.

Mapping the loop bounds requires a four-step algorithm that is shown in Figure 2.15. The first and second steps are to extract inequalities, minima, and maxima from the initial source code. Step 3 is transforming the indices, given the transformation matrix T. Also, the inequalities, minima, and maxima are transformed based upon the new loop indices. Finally, the new loop bounds are calculated based upon the inequalities, minima, and maxima determined in Step 3.

Algorithm map_bounds
Maps the loop bounds from the initial index variables $(I_1, \ldots, I_n)$ to the transformed index variables $(J_1, \ldots, J_n)$.

Input:

$I_1, \ldots, I_n$	/***the initial index variables***/
B	/***the loop bound set for the initial index variables***/
T	/***the transformation matrix***/

Output:

$J_1, \ldots, J_n$	/***the transformed index variables***/
B'	/***the loop bound set for the transformed index variables***/

Procedure:

1. Extract inequalities.
   /***Inequalities are derived from the upper and lower bounds of each loop index variable.***/

2. Find the absolute minimum and maximum for each initial index variable.
   /***Minima and maxima for each index variable are based upon the inequalities from Step 1.***/

3. Transform the indices using T.
   /***The transformation matrix T is used to map the initial index variables into transformed index variables. Also, the transformed index variables are substituted into the inequalities, maxima, and minima calculated in Steps 1 and 2.***/

4. Calculate the new loop bounds.
   /***The new loop bounds for the final index variables are determined based upon the inequalities and minima and maxima from Step 3.***/

Figure 2.15. Algorithm to map the loop bounds.

The loop bounds for the source code shown in Figure 2.10 are mapped as shown in Figure 2.16, assuming that the lower bounds for all loops is one ($\forall i : 1 \leq i \leq 4 : lb_i = 1$).

The boxed expressions represent the algebraic solutions to the inequalities in terms of the transformed index variables. The transformed inequalities are used to determine the loop bounds of the transformed index variables $(J_1, \ldots, J_4)$.

Step 1. Extract inequalities

$I_1 \geq 1 \quad I_2 \geq 1 \quad I_3 \geq 1 \quad I_4 \geq 1$
$I_1 \leq ub_1 \quad I_2 \leq ub_2 \quad I_3 \leq ub_3 \quad I_4 \leq ub_4$

Step 2. Find maxima and minima

$\min(I_1) = 1 \quad \min(I_2) = 1 \quad \min(I_3) = 1 \quad \min(I_4) = 1$
$\max(I_1) = ub_1 \quad \max(I_2) = ub_2 \quad \max(I_3) = ub_3 \quad \max(I_4) = ub_4$

Step 3. Transform the indices

Using results from assignment statement mapping.

$I_1 = J_2$	$I_2 = J_1 - 2j_2$	$I_3 = -J_3$	$I_4 = J_4$
$J_2 \geq 1$	$J_1 - 2J_2 \geq 1$	$-J_3 \geq 1$	$J_4 \geq 1$
$J_2 \leq ub_1$	$J_1 - 2J_2 \leq ub_2$	$-J_3 \leq ub_3$	$J_4 \leq ub_4$
$\min(J_2) = 1$	$\min(J_1 - 2j_2) = 1$	$\min(-J_3) = 1$	$\min(J_4) = 1$
$\max(J_2) = ub_1$	$\max(J_1 - 2j_2) = ub_2$	$\max(-J_3) = ub_3$	$\max(J_4) = ub_4$

Step 4. Calculate new loop bounds

$\boxed{J_2 \geq 1}$
$\boxed{J_2 \leq ub_1}$

$J_1 - 2J_2 \geq 1$
$J_1 \geq 1 + 2J_2$
$J_1 \geq 1 + 2\min(J_2)$
$\boxed{J_1 \geq 3}$

$-J_3 \geq 1$
$\boxed{J_3 \leq -1}$

$-J_3 \leq ub_3$
$\boxed{J_3 \geq -ub_3}$

$\boxed{J_4 \geq 1}$
$\boxed{J_4 \leq ub_4}$

$J_1 - 2J_2 \leq ub_2$
$J_1 \leq ub_2 + 2J_2$
$J_1 \leq ub_2 + 2\max(J_2)$
$\boxed{J_1 \leq ub_2 + 2ub_1}$

Final code

```
L1  for J1 = 3, ub2 + 2ub1
L2    for J2 = 1, ub1
L3      for J3 = -ub3, -1
L4        for J4 = 1, ub4
S1          A[J2, J1 - 2J2, -J3, J4] = ...
S2          A[J2 - 1, J1 - 2J2 - 3, -J3 + 2, J4] = ...
```

Figure 2.16. Transformation of loop bounds.

The complexity of the source code transformation is dependent upon the loop nest depth n, the number of assignment statements m, and the number of inequalities derived from the source code q.

Mapping assignment statements has the following complexity:

- matrix multiply $O(n^2)$.
- search and replace occurrences $O(n(n+m)) = O(n^2+nm)$. Assumes that there is an index variable for each loop.
- Total $O(n^2 + nm) = O(n^2)$.

Mapping the loop bounds has the following complexity:

Step 1. Extract inequalities $O(n)$.

Step 2. Find minima and maxima $O(n)$.

Step 3. Transform loop bound indices $O(n^2 + q)$. Matrix multiply and scan through inequalities.

Step 4. Calculate new loop bounds $O(nq)$. For each index variable, search through set of inequalities.

Total $O(nq)$. There are always at least two inequalities from each loop.

Note that the code is always transformed using this method. Thus, the average and worst case complexity are equivalent. Also, the code is transformed only once, after the final transformation code T is found.

CHAPTER 3

Kernel Transformations and Loop Normalization

The chapter briefly describes how the kernel set of loop transformations are applied to imperfect loop nests. This chapter also describes loop normalization, a transformation that modifies a loop L_i, such that the lower bound lb_i, and step size s_i, are both one.

3.1 Kernel Transformations

The kernel set of transformations includes loop reversal, permutation, and skewing. The legality and applicability of the kernel set is well understood in the context of perfectly nested loops. This section discusses the legality of the kernel set in terms of imperfect-only-child loop nests. Transformation matrices cannot be applied to imperfect-sibling loop nests; however, many imperfect-sibling loop nests can be transformed into imperfect-only-child loop nests by loop fission (Section 5.1).

For simplicity, a doubly-nested loop is used in the following discussion. However, the concepts can be easily generalized to an n-nested loop, where $n > 2$. The general form of an imperfect-only-child loop nest is shown in Figure 3.1, where $f(), g()$, and $h()$ are functions of the enclosing index variables.

In the following discussion, the predicate $normal(B^{(x)}, i)$ is true if loop L_i is normal in loop bound set $B^{(x)}$. A loop is normal if both the lower bound and step size are one.

```
L1  for I1 = lb1, ub1, s1
S1   ARRAY[f(I1)] = ...
L2   for I2 = lb2, ub2, s2
S2    ARRAY[g(I1, I2)] = ...
     endfor
S3   ARRAY[h(I1)] = ...
     endfor
```

Figure 3.1. Sample code: general form of an imperfectly nested loop.

3.1.1 Loop Reversal

Loop reversal of loop L_i results in a modified loop and all references to the index variable I_i being negated. Specifically, if the initial loop is incremented from 1 to ub_i, then the reversed loop increments from $-ub_i$ to -1. Loop reversal is applied to a single loop. Reversal of the outer loop (L_1) of the source code shown in Figure 3.1 results in the source code shown in Figure 3.2(a), and reversal of the inner loop (L_2) results in the source code shown in Figure 3.2(b).

```
L1  for I1 = -ub1, -lb1, s1
S1   ARRAY[f(-I1)] = ...
L2   for I2 = lb2, ub2, s2
S2    ARRAY[g(-I1, I2)] = ...
     endfor
S3   ARRAY[h(-I1)] = ...
     endfor
```

Figure 3.2(a). Sample code: reversal of loop L_1.

```
L1  for I1 = lb1, ub1, s1
S1   ARRAY[f(I1)] = ...
L2   for I2 = -ub2, -lb2, s2
S2    ARRAY[g(I1, -I2)] = ...
     endfor
S3   ARRAY[h(I1)] = ...
     endfor
```

Figure 3.2(b). Sample code: reversal of loop L_2.

As shown in Figures 3.2(a) and 3.2(b), the assignment statements S_1, S_2, and S_3 have the same number of iterations before and after loop reversal. The loop bounds and corresponding references to the index variables are reversed; however, the source code is semantically equivalent (assuming that the global postcondition is true).

Loop reversal is always legal for imperfect-only-child loops. That is, loop reversal can be applied to source code regardless of whether or not it is perfectly nested. All kernel

unimodular transformations require that the step size be one prior to transformation. The invariant below corresponds to this requirement.

Preconditions

Invariant

$\boxed{normal(B^{(x-1)}, i)}$

Explicit

$\boxed{\text{TRUE}}$

Nesting

$\boxed{\text{TRUE}}$

3.1.2 Loop Permutation

Loop permutation of loops L_i and L_j results in the interchange of the lower bounds, upper bounds, and step sizes for the two loops. Figure 3.3 shows the permuted source code from Figure 3.1. The arrays in S_1 and S_3 are functions of the index variable of loop L_1. After loop interchange, loop L_1 and index variable I_1, are inside of statements S_1 and S_3, and the functions $f(I_1)$ and $h(I_1)$ have no context. That is, an assignment statement that uses an index variable to calculate a subscript expression must be contained by the loop that increments the index variable.

```
L2   for I2 = lb2, ub2, s2
S1    ARRAY[f(I1)] = ...
L1    for I1 = lb1, ub1, s1
S2     ARRAY[g(I1, I2)] = ...
      endfor
S3    ARRAY[h(I1)] = ...
     endfor
```

Figure 3.3. Sample code: illegal permutation of loops L_1 and L_2.

Loop permutation may result in loop L_i following the assignment statement that references it. Therefore, if two loops L_i and L_j are permuted, then all parent-child pairs of

loops in between loops L_i and L_j must be perfectly nested. Again, the invariant for loop permutation is that both loops L_i and L_j are normal. Finally, we assume that loop L_i is initially outermost when loops L_i and L_j are interchanged.

Preconditions

Invariant

$$\boxed{normal(B^{(x-1)}, i) \wedge normal(B^{(x-1)}, j)}$$

Explicit

$$\boxed{i < j}$$

Nesting

$$\boxed{(\forall k : i \leq k < j : perfect(L_k, L_{k+1}))}$$

where $perfect(L_i, L_{i+1})$ is *true* if loops L_i and L_{i+1} are perfectly nested in each other.

3.1.3 Loop Skewing

Loop skewing results in the modification of the loop bounds and references to an index variable I_j with respect to an outer index variable I_i. The upper bound and lower bound of loop L_j are increased by a linear function m of index variable I_i, and all references to index variable I_i inside of the skewed loop are decreased by $m \times I_i$. As an example, skewing I_2 by m times I_1 results in the source code shown in Figure 3.4.

```
L1   for I1 = lb1, ub1, s1
S1    ARRAY[f(I1)] = ...
L2    for I2 = lb2 + mI1, ub2 + mI1, s2
S2     ARRAY[g(I1, I2 - mI1)] = ...
      endfor
S3    ARRAY[h(I1)] = ...
      endfor
```

Figure 3.4. Sample code: skew of loop L_2 by $m \times L_1$.

Perhaps surprisingly, loop skewing is like loop reversal in that it can be applied to a target source code regardless of the nesting structure of the code. Loop permutation results

in two loops being modified. Loop skewing is like loop reversal because only one loop is modified by a function of an enclosing loop. Therefore, the *nesting precondition* is *true*. We assume that loop L_i is outermost and the invariant is that both loops are normal.

Preconditions

Invariant

$$\boxed{normal(B^{(x-1)}, i) \wedge normal(B^{(x-1)}, j)}$$

Explicit

$$\boxed{i < j}$$

Nesting

$$\boxed{\text{TRUE}}$$

3.2 Loop Normalization

We have developed a matrix-based representation of the *loop normalization* transformation [13, 14, 15]. Loop normalization is a transformation that changes the bounds of a loop, such that the lower bound and step size are both one. The transformation is important because normalized loops are a precondition for many kernel and extended loop transformations. For example, the step size of a loop must equal one before any kernel transformations are applied, and fully normalized loops make loop blocking and collapsing easier to implement. Also, loop normalization may decrease the cardinality of the distance vector set D, resulting in greater opportunities for parallelism.

An example of code that is not normalized is shown in Figure 3.5(a), where neither the lower bounds nor the step sizes are equal to one for loops L_1 and L_2. In general, loop normalization of loop L_i results in ub_i being replaced with $\lfloor (ub_i - lb_i + s_i)/s_i \rfloor$ and all occurrences of index variable I_i being replaced by $lb_i + (J_i - 1)s_i$. In Figure 3.5(b), loop normalization is applied to the outermost loop (L_1), and in Figure 3.5(c), loop normalization is applied to the innermost loop (L_2).

In order to normalize the outer loop in Figure 3.5(a) the upper bound of loop L_1 is replaced with $\lfloor (ub_{I_1} - lb_{I_1} + s_{I_1})/s_{I_1} \rfloor$. As shown in Figure 3.5(b), I_1 is renamed J_1 and the

upper bound of J_1 is calculated as:

$$ub_{J_1} = \lfloor (ub_{I_1} - lb_{I_1} + s_{I_1})/s_{I_1} \rfloor = \lfloor (13 - 2 + 3)/3 \rfloor = \lfloor 14/3 \rfloor = 4.$$

All occurrences of I_1 are replaced with $lb_{I_1} + (J_1 - 1)s_{I_1}$:

$$I_1 \Rightarrow lb_{I_1} + (J_1 - 1)s_{I_1} = 2 + (J_1 - 1)3 = 3J_1 - 1.$$

Note that occurrences of I_1 in the loop bounds of loop L_2, as well as in assignment statement S_1, are replaced with $3J_1 - 1$.

Similarly, the innermost loop (L_2) is normalized as shown in Figure 3.5(c), where the upper bound is calculated as:

$$ub_{J_2} = \lfloor (ub_{I_2} - lb_{I_2} + s_{I_2})/s_{I_2} \rfloor = \lfloor (10 + 6J_1 - (3J_1 - 1) + 6)/6 \rfloor = \lfloor (3J_1 + 17)/6 \rfloor,$$

and occurrences of I_2 in the source code are replaced with:

$$I_2 \Rightarrow lb_{I_2} + (J_2 - 1)s_{I_2} = 3J_1 - 1 + (J_2 - 1)6 = 3J_1 + 6J_2 - 7.$$

```
L1  for I1 = 2,13,3
L2    for I2 = I1,12+2I1,6
S1      A[I1,I2] = ...
```

Figure 3.5(a). Sample code: before loop normalization.

```
L1  for J1 = 1,4
L2    for I2 = 3J1-1,10+6J1,6
S1      A[3J1-1,I2] = ...
```

Figure 3.5(b). Sample code: loop normalization of loop L_1.

```
L1  for J1 = 1,4
L2    for J2 = 1,⌊(3J1+17)/6⌋
S1      A[3J1-1,
          3J1+6J2-7] = ...
```

Figure 3.5(c). Sample code: loop normalization of loop L_2.

3.2.1 Legality Criteria

Loop normalization is always legal, although it may not be necessary. That is, loop L_i may already be normal ($lb_i = 1$ and $s_i = 1$) and, therefore, loop normalization does not

modify the source code. However, if the application of a kernel or extended transformation results in unnormalized code, then loop normalization typically follows in the transformation sequence.

The predicate $normal(B^{(x)}, i)$ is true if loop L_i is normal in loop bound set $B^{(x)}$. The explicit precondition is that the loop is not normal before loop normalization is applied.

Preconditions

Invariant

$\boxed{\text{TRUE}}$

Explicit

$\boxed{\neg(normal(B^{(x-1)}, i))}$

Nesting

$\boxed{\text{TRUE}}$

3.2.2 Effect on the Dependence Vector Set D

If loop L_i is not normal, then it is in one of three forms. Either the step size of loop L_i is not one, the lower bound of loop L_i is not one, or neither the step size nor the lower bound of loop L_i is one. Figure 3.6 shows each of these three cases: loop L_1 fits case 1 because the step size does not equal one ($s_i \neq 1 \wedge lb_i = 1$); loop L_2 fits case 2 because the lower bound does not equal one ($s_i = 1 \wedge lb_i \neq 1$); and loop L_3 fits case 3 because neither the step size nor the lower bound is one ($s_i \neq 1 \wedge lb_i \neq 1$). Each case is more thoroughly described below.

Case 1. $s_i \neq 1 \wedge lb_i = 1$. Loop normalization results in modification to all distance vectors in the distance vector set. Specifically, the distances at location i (d_i) are divided by the current step size s_i. Intuitively, the modification to the distances is logical since the distances between executions of assignment statements are reduced by $\frac{1}{s_i}$. Formally, if loop normalization is the x^{th} trans-

```
L1   for I1 = 1, m, 2
L2    for I2 = 3I1, n
L3     for I3 = 2I2, p, 5
S1      A[I1, I2, I3] = ...
S2      A[I1 − 4, I2 + 5, I3 − 10] = ...

/**D = {(4, −5, 10)}**/
```

Figure 3.6. Sample code: different cases of unnormal loops.

formation, then the modification to the distance vector set D is described as:

$$(\forall \vec{d}^{(x-1)} : \vec{d}^{(x-1)} \in D^{(x-1)} : d_i^{(x)} = \frac{d_i^{(x-1)}}{s_i}).$$

If any distances in $\vec{d}^{(x)}$ are non-integral, then the distance vector is removed from the distance vector set.

The effect on D is represented by the transformation matrix T. Suppose that $T^{(x-1)}$ is the transformation matrix before loop normalization and t_n is the elementary matrix representing normalization. Then $T^{(x)} = t_n \times T^{(x-1)}$ becomes the new transformation matrix, where t_n is the identity matrix except that $t_n[i,i] = \frac{1}{s_i}$.

As an example, the outermost loop L_1 in Figure 3.6 is not normal. In order to normalize L_1, $t_n[1,1]$ must equal $\frac{1}{s_1}$, or $\frac{1}{2}$. Therefore, the t_n that normalizes L_1 is:

$$t_n = \begin{bmatrix} \frac{1}{s_i} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and the modified distance vector set is $t_n \times \vec{d} \in D$:

$$\begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 4 \\ -5 \\ 10 \end{bmatrix} = \begin{bmatrix} 2 \\ -5 \\ 10 \end{bmatrix}$$

Case 2. $s_i = 1 \wedge lb_i \neq 1$. Again, loop normalization of L_i results in modification to all $\vec{d}$. The assumption is made that lb_i is constant, or it is a linear function of an outer loop. Let $a_{i,j}$ be the integer coefficient of the index variable I_j with respect to the lower bound of loop L_i. As an example, $a_{2,1} = 3$ for loop L_2 of the source code shown in Figure 3.6. In order to normalize loop L_2, the lower bound of loop L_2 (lb_2) must be modified to equal one. Therefore, a skew is required, and the skew factor is equal to the negation of the coefficient $a_{i,j}$. In the case of loop L_2 in Figure 3.6, the skew factor is $-a_{2,1}$ or -3. Also, since lb_2 ($i = 2$) is skewed with respect to index variable I_1 ($j = 1$), the location of the skew is $t_n[i,j] = t_n[2,1]$:

$$t_n = \begin{bmatrix} 1 & 0 & 0 \\ -a_{2,1} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If loop L_i fits Case 2, then the form of the lower bound of loop L_i is $lb_i = a_{i,j} \times I_j$. That is, the lower bound is a linear function of the index variable of an outer loop. The distances at location i ($d_i^{(x-1)}$) are modified by subtracting the coefficient $a_{i,j}$ times the distance at j ($d_j^{(x-1)}$), as in loop skewing. Formally, this condition is described as:

$$(\forall \vec{d}^{(x-1)} : \vec{d}^{(x-1)} \in D^{(x-1)} : d_i^{(x)} = d_i^{(x-1)} - (a_{i,j} \times d_j^{(x-1)})).$$

Using the distance vector from the source code in Figure 3.6, $\vec{d}^{(x)} = t_n \times \vec{d}^{(x-1)}$:

$$\begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 4 \\ -5 \\ 10 \end{bmatrix} = \begin{bmatrix} 4 \\ -17 \\ 10 \end{bmatrix}$$

Case 3. $s_i \neq 1 \wedge lb_i \neq 1$. In Case 3, neither the step size nor the lower bound of loop L_i equals one, and, not surprisingly, the modifications to D reflect both normalization of the step size to equal one (Case 1) and the lower bound to equal one (Case 2). First, we consider the lower bound lb_i. As was shown in Case 2, $lb_i = a_{i,j} \times I_j$, where $j < i$. The matrix that normalizes the lower bound is t_n such that $t_n[i,j] = -a_{i,j}$. In Case 1, the step size of loop L_i was not one ($s_i \neq 1$), and the matrix to normalize the loop was t_n such that $t_n[i,i] = \frac{1}{s_i}$. If both Cases 1 and 2 are true, then $t_n[i,j]$ is $-a_{i,j}$, and then the entire i^{th} row is multiplied by $\frac{1}{s_i}$.

As an example, loop L_3 in Figure 3.6 has a lower bound that is a linear function of I_2 and the step size does not equal one. That is, $lb_3 = a_{3,j} \times I_j = 2 \times I_2$ ($i = 3, j = 2, a_{3,2} = 2$) and $s_i = 5$. Therefore, $t_n[i,j] = t_n[3,2] = -2$ and the $i^{th}(3^{rd})$ row is multiplied by $\frac{1}{s_i} = \frac{1}{s_3} = \frac{1}{5}$.

The elementary transformation matrix t_n is:

$$t_n = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{a_{3,2}}{s_3} & \frac{1}{s_3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{2}{5} & \frac{1}{5} \end{bmatrix}$$

Without loss of generality, the form of t_n for normalization of all loops in a 3-nested loop is:

$$t_n = \begin{bmatrix} \frac{1}{s_1} & 0 & 0 \\ -\frac{a_{2,1}}{s_2} & \frac{1}{s_2} & 0 \\ -\frac{a_{3,1}}{s_3} & -\frac{a_{3,2}}{s_3} & \frac{1}{s_3} \end{bmatrix}$$

where $a_{i,j}$ is the appropriate coefficient and s_i is the step size for loop L_i. The matrix t_n can be easily generalized to $n > 3$, where n is the loop nest depth.

Another advantage of loop normalization is that it discriminates between many actual dependences and "apparent" dependences. Dependence tests make conservative assumptions when calculating dependences for source code. That is, if calculating the distances is too complex, then a dependence test *assumes* that a dependence exists. Also, more dependence vectors in D lead directly to more restrictions on the amount of attainable parallelism. Therefore, it is desirable that the distance vector set D has minimal cardinality and accurate distance vectors. Loop normalization allows the elimination of distance vectors that are erroneously in D as a result of a conservative assumption by a dependence test.

As an example, the source code in Figure 3.7 has an apparent distance vector set of $D = \{(8,2,5)\}$. The transformation matrix that represents loop normalization of loops L_1, L_2, and L_3 is:

$$\begin{bmatrix} \frac{1}{4} & 0 & 0 \\ \frac{-2}{2} & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \times \begin{bmatrix} 8 \\ 2 \\ 5 \end{bmatrix} = \begin{bmatrix} 1 \\ -7 \\ \frac{5}{3} \end{bmatrix}$$

After normalization, one of the distances in the distance vector set is non-integral ($\frac{5}{3}$). Since all distances in a distance vector must be integral, the distance vector $\vec{d} = \{(8,2,5)\}$ is eliminated from the distance vector set, removing potential restrictions on available parallelism in the source code.

```
L1  for i = 3, m, 4
L2    for j = 2 + 2i, n + 2i, 2
L3      for k = -p, -1, 3
S1        A[i, j, k] = ...
S2        A[i - 8, j - 2, k - 5] = ...

/**D = {(8, 2, 5)}**/
```

Figure 3.7. Sample code: false dependence.

In summary, an elementary matrix t_n is found that represents the effect of loop normalization of the distance vector set. The matrix t_n is used to represent normalization of the step size, lower bound, or both, of any loop. The matrix t_n composes with other kernel and extended transformations. Finally, loop normalization helps to eliminate "apparent" distance vectors in the distance vector set, thus increasing the amount of potential parallelism in the transformed source code.

3.2.3 Effect on Source Code Mapping

The general form of a doubly-nested loop before loop normalization is shown in Figure 3.8(a). Normalization of the outer loop L_1 results in a new upper bound:

$$ub_{J_1} = \lfloor (ub_1 - lb_1 + s_1)/s_1 \rfloor = \lfloor (d_1 - c_1 + s_1)/s_1 \rfloor,$$

and any occurrences of I_1 are replaced with:

$$lb_1 + (J_1 - 1)s_1 = c_1 + J_1 s_1 - s_1. \tag{3.1}$$

Similarly, normalization of the inner loop results in the upper bound of loop L_2 becoming:

$$ub_{J_2} = \lfloor (ub_2 - lb_2 + s_2)/s_2 \rfloor = \lfloor ((b_{2,1}(c_1 + J_1 s_1 - s_1) + d_2) - (a_{2,1}(c_1 + J_1 s_1 - s_1) + c_2) + s_2)/s_2 \rfloor$$

and any occurrences of I_2 are replaced with:

$$lb_2 + (J_2 - 1)s_2 = (a_{2,1}(c_1 + J_1 s_1 - s_1) + c_2) + J_2 s_2 - s_2. \tag{3.2}$$

Source code with both L_1 and L_2 normalized is shown in Figure 3.8(b).

The original assignment statement S_1 was $\mathtt{A}[I_1, I_2]$ in Figure 3.8(a), but after substitution using Equations (3.1) and (3.2) becomes:

$$\mathtt{A}[c_1 + J_1 s_1 - s_1, a_{2,1}(c_1 + J_1 s_1 - s_1) + c_2 + J_2 s_2 - s_2]. \tag{3.3}$$

```
L1   for I1 = c1, d1, s1
L2    for I2 = a2,1 I1 + c2, b2,1 I1 + d2, s2
S1     A[I1, I2] = ...
```

Figure 3.8(a). General form of unnormalized double-nested loop.

```
L1   for J1 = 1, ⌊(d1 − c1 + s1)/s1⌋
L2    for J2 = 1, ⌊((b2,1(c1 + J1 s1 − s1) + d2)
                 −(a2,1(c1 + J1 s1 − s1) + c2) + s2)/s2⌋
S1     A[c1 + J1 s1 − s1,
         a2,1(c1 + J1 s1 − s1) + c2 + J2 s2 − s2] = ...
```

Figure 3.8(b). General form of normalized double-nested loop.

Therefore, after loop normalization of loops L_1 and L_2, we know that:

$$I_1 = c_1 + J_1 s_1 - s_1, \tag{3.4}$$

and

$$I_2 = a_{2,1}(c_1 + J_1 s_1 - s_1) + c_2 + J_2 s_2 - s_2. \tag{3.5}$$

Manipulating Equation (3.4), and placing it in terms of J_1, we obtain:

$$J_1 = \frac{I_1}{s_1} - \frac{c_1}{s_1} + \frac{s_1}{s_1}. \tag{3.6}$$

Similarly, putting Equation (3.5) in terms of J_2, we obtain:

$$J_2 = \frac{I_2}{s_2} - \frac{a_{2,1} I_1}{s_2} - \frac{c_2}{s_2} + \frac{s_2}{s_2}. \tag{3.7}$$

Recall from Section 2.7, that transforming source code involves the matrix representation of the new index variables in terms of the initial index variables. In matrix form, the representation is $[J_i] = [T] \times [I_i]$, where J_i are the new index variables, T is the transformation matrix, and I_i are the initial index variables.

The transformation matrix T after loop normalization of loops L_1 and L_2 in Figure 3.8(a) is in the form:

$$\begin{bmatrix} \frac{1}{s_1} & 0 \\ -\frac{a_{2,1}}{s_2} & \frac{1}{s_2} \end{bmatrix} \tag{3.8}$$

Putting Equations (3.6) and (3.7) in matrix form results in:

$$\begin{bmatrix} J_1 \\ J_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{s_1} & 0 \\ -\frac{a_{2,1}}{s_2} & \frac{1}{s_2} \end{bmatrix} \times \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} - \underbrace{\begin{bmatrix} \frac{c_1 - s_1}{s_1} \\ \frac{c_2 - s_2}{s_2} \end{bmatrix}}_{\text{additional terms}} \tag{3.9}$$

The form of the matrix representation of Equations (3.6) and (3.7) is consistent with the kernel unimodular transformations ($[J_i] = [T][I_i]$). However, an additional term is subtracted from each new index variable which represents the effect of normalization. The additional terms from Equation (3.9) are rewritten in terms of the normalization matrix t_n:

$$\begin{bmatrix} \frac{c_1 - s_1}{s_1} \\ \frac{c_2 - s_2}{s_2} \end{bmatrix} = \begin{bmatrix} \frac{1}{s_1} & 0 \\ -\frac{a_{2,1}}{s_2} & \frac{1}{s_2} \end{bmatrix} \times \begin{bmatrix} c_1 - s_1 \\ a_{2,1}(c_1 - s_1) + c_2 - s_2 \end{bmatrix} \tag{3.10}$$

Substituting Equation (3.10) into Equation (3.9) results in the matrix representation:

$$\begin{bmatrix} J_1 \\ J_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{s_1} & 0 \\ -\frac{a_{2,1}}{s_2} & \frac{1}{s_2} \end{bmatrix} \times \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} - \begin{bmatrix} \frac{1}{s_1} & 0 \\ -\frac{a_{2,1}}{s_2} & \frac{1}{s_2} \end{bmatrix} \times \begin{bmatrix} c_1 - s_1 \\ a_{2,1}(c_1 - s_1) + c_2 - s_2 \end{bmatrix} \tag{3.11}$$

The remainder of the source code mapping process from initial to final form, given a transformation matrix T, is the same after loop normalization as it was after only kernel transformations. That is, the four-step approach described in Section 2.7 is directly applied for source code mapping.

In summary, the general form of the index variable mapping of a normalized loop is:

$$\begin{bmatrix} J_1 \\ J_2 \\ \vdots \\ J_n \end{bmatrix} = T \times \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_n \end{bmatrix} - T \times \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix}$$

where each J_i is a normalized index variable, each I_i is an initial index variable, T is the transformation matrix, and each V_i is a normalization expression that is a function of the coefficients and step sizes of the initial loops.

CHAPTER 4

Loop Blocking and Loop Collapsing

Loop blocking is the generic name for a family of transformations that increase the depth of a loop nest, and includes tiling, strip mining, and cycle shrinking. *Loop collapsing* is the inverse of loop blocking and decreases the depth of a loop nest. Repeated application of loop collapsing is called loop coalescing. This chapter describes the matrix-based representations of both loop blocking and loop collapsing [12, 23].

4.1 Loop Blocking

Loop blocking is the name given to a family of loop transformations that increase the nest depth of a nested loop. The objective of increasing the nest depth is to divide the initial iteration space into several iteration subspaces that are distributed and executed in parallel. *Strip mining* [4, 16, 17, 18, 19], *cycle shrinking*, and *tiling* [4, 19] are all generalized as one transformation in the extended matrix-based model presented here. Loop blocking represents each of these individual transformations, dependent upon the quantity of loops that are blocked and which loops are parallelized.

Strip mining is a technique that transforms an n-nested loop into an $n+1$-nested loop. The n^{th} loop is divided into equal size partitions (s_n) and is executed in parallel. Strip mining is referred to as a *memory optimizing transformation* [17] because s_n is dependent

upon some feature of the machine architecture. An advantage of strip mining is that fine-grained parallelism is obtained by parallelizing the n^{th} loop.

Cycle shrinking is similar to strip mining, since an n-nested loop is transformed into an $n+1$-nested loop. However, in cycle shrinking, s_n is dependent upon the source code, rather than the machine. Also, the $n+1^{st}$ loop, rather than the n^{th} loop, is parallelized. An advantage of cycle shrinking is that each iteration within the $n+1^{st}$ loop may be executed in parallel, as long as the initial dependence distances of the n^{th} loop are greater than one. Therefore, a fine-grain intra-block parallelism is obtained.

Finally, tiling is a transformation that divides the initial iteration space into some fixed number of subspaces, or *tiles*. Each original loop may be partitioned based upon an architectural characteristic (as in strip mining) or a source code characteristic (as in cycle shrinking). Using tiling, an n-nested loop is transformed into a loop with a nest depth between $n+1$ and $2n$. As an example, the 3-nested loop in Figure 4.1(a) becomes the 6-nested loop given in Figure 4.1(b) if each loop is tiled. An advantage of tiling is that the transformed source code can be evaluated for parallelism within tiles (intra-tile parallelism) and between tiles (inter-tile parallelism). Thus, tiling can be used to obtain both fine-grain and coarse-grain parallelism.

L_1 **for** $I_1 = 1, n$
L_2 **for** $I_2 = 1, m$
L_3 **for** $I_3 = 1, p$
S_1 $A[I_1, I_2, I_3] = \ldots$
S_2 $B[I_1 - 2, I_2 - 1, I_3 - 1] = \ldots$

Figure 4.1(a). Sample code: before transformation.

L'_1 **for** $I'_1 = 1, n, s_1$
L'_2 **for** $I'_2 = 1, m, s_2$
L'_3 **for** $I'_3 = 1, p, s_3$
L_1 **for** $I_1 = I'_1, min(I'_1 + s_1 - 1, n)$
L_2 **for** $I_2 = I'_2, min(I'_2 + s_2 - 1, m)$
L_3 **for** $I_3 = I'_3, min(I'_3 + s_3 - 1, p)$
S_1 $A[I_1, I_2, I_3] = \ldots$
S_2 $B[I_1 - 2, I_2 - 1, I_3 - 1] = \ldots$

Figure 4.1(b). Sample code: after *loop blocking* all loops.

Traditionally, loop blocking and tiling are considered synonymous, and strip mining and cycle shrinking are considered subclasses. That is, loop blocking typically refers to tiling a *range* of loops (L_i through L_j) within a loop nest and strip mining and cycle shrinking refer to transforming a single loop (L_i). In this dissertation, blocking is applied to a single loop as in strip mining. Tiling is equivalent to a combination of blocking and permutation using this definition of blocking.

As an example, to tile the source code shown in Figure 4.1(a) we would invoke the function **tile**$(L_i, L_j, b[i, \ldots, j])$. Loops L_i through L_j are tiled with blocking factors of b_i through b_j, respectively. The resulting source code is shown in Figure 4.1(b).

In contrast, obtaining the source code shown in Figure 4.1(b) is a six-step process using the primitive **block** and **permute**:

1. **block**(L_1, s_1);
2. **block**(L_2, s_2);
3. **block**(L_3, s_3);
4. **permute**(L_1, L_2');
5. **permute**(L_1, L_3');
6. **permute**(L_1, L_2).

First, the outer loop L_1 is blocked with blocking factor s_1 (Figure 4.2(a)). Next, loops L_2 and L_3 are blocked with blocking factors s_2 and s_3, respectively (Figure 4.2(b)). As is shown in Figure 4.2(b), blocking of all three loops results in the blocking loop L_i' directly enclosing the blocked loop L_i. Finally, permutation is used to move all of the blocking loops outermost and all of the blocked loops innermost. Figure 4.2(c) shows the transformed source code just prior to the final permutation.

Although redefining *blocking* seems like a complication, it is actually a simplification. The composed transformation *tiling* is subdivided into its constituent elementary transformations. Rather than discussing the legality of a composite transformation, the legality of the constituent transformations are further explored and more easily understood.

```
L'1   for I'1 = 1, n, s1
L1     for I1 = I'1, min(n, I'1 + s1 - 1)
L2      for I2 = 1, m
L3       for I3 = 1, p
S1        A[I1, I2, I3] = ...
S2        B[I1 - 2, I2 - 1, I3 - 1] = ...
```

Figure 4.2(a). Sample code: after blocking loop L_1.

```
L'1   for I'1 = 1, n, s1
L1     for I1 = I'1, min(n, I'1 + s1 - 1)
L'2     for I'2 = 1, m, s2
L2       for I2 = I'2, min(m, I'2 + s2 - 1)
L'3       for I'3 = 1, p, s3
L3         for I3 = I'3, min(p, I'3 + s3 - 1)
S1          A[I1, I2, I3] = ...
S2          B[I1 - 2, I2 - 1, I3 - 1] = ...
```

Figure 4.2(b). Sample code: after blocking loops L_1 through L_3.

```
L'1   for I'1 = 1, n, s1
L'2    for I'2 = 1, m, s2
L'3     for I'3 = 1, p, s3
L2       for I2 = I'2, min(m, I'2 + s2 - 1)
L1        for I1 = I'1, min(n, I'1 + s1 - 1)
L3         for I3 = I'3, min(p, I'3 + s3 - 1)
S1          A[I1, I2, I3] = ...
S2          B[I1 - 2, I2 - 1, I3 - 1] = ...
```

Figure 4.2(c). Sample code: before permuting loops L_2 and L_1.

4.1.1 Legality Criteria

Historically, tiling loops L_i through L_j was legal if they were fully permutable [24]. However, tiling involves blocking single loops and then permuting the blocking loops to the outermost position and the blocked loops to the innermost position. The precondition of full permutability is necessary to allow the second stage (permutation) of the loops.

However, in this dissertation the transformations of loop blocking and loop permutation are separated. The precondition of full permutability is only relevant in the context of the kernel transformation of loop permutation. As an example, the source code in Figure 4.2(b) shows the blocking of each loop L_1 through L_3, without any permutation. Loop blocking, as an elemental transformation, is always legal as long as the bounds are normal

prior to the transformation. It is assumed that loop normalization follows loop blocking in the transformation sequence and returns the source code to normal form.

Preconditions

Invariant

$\boxed{normal(B^{(x-1)}, i)}$

Explicit

$\boxed{\text{TRUE}}$

Nesting

$\boxed{\text{TRUE}}$

4.1.2 Effect on the Dependence Vector Set D

Given that the loop nest depth is n, and loop blocking is the x^{th} transformation in the sequence, then all distance vectors prior to loop blocking have a length n ($\forall \vec{d}^{(x-1)} : \vec{d}^{(x-1)} \in D^{(x-1)} : |\vec{d}^{(x-1)}| = n$). Blocking a loop increases the nest depth by one and also increases the length of each distance vector by one ($\forall \vec{d}^{(x)} : \vec{d}^{(x)} \in D^{(x)} : |\vec{d}^{(x)}| = n + 1$). As can be seen, blocking a range of loops L_i through L_j results in all distance vectors becoming n' long, where $n' = n + (j - i + 1)$.

Loop blocking may increase the cardinality of the distance vector set D, based upon the values calculated from the distances before loop blocking and the block size. Specifically, if $\vec{d}^{(x-1)} \in D^{(x-1)}$ are the distance vectors before loop blocking, and loop L_i is blocked, then:

- distances in loops outside of the blocked code ($1 \leq k < i$) remain unchanged;
- distances at loop L_i ($k = i$) are assigned values calculated from the block size and the distance before the loop blocking transformation;
- and distances in loops inside of the blocked code ($i < k \leq n + 1$) are assigned the original distances with a location offset by one.

Formally, the mapping rule from $D^{(x-1)}$ to $D^{(x)}$, where loop blocking is the x^{th} transformation and loop L_i is blocked with the blocking factor s_i:

$$
\begin{aligned}
&(\forall \vec{d} : \vec{d} \in D^{(x-1)} : (\lceil d_i/s_i \rceil = \lfloor d_i/s_i \rfloor \wedge \\
&\quad (\forall d_k : (1 \le k < i : d'_k = d_k) \vee \\
&\qquad (k = i : d'_k = d_i) \vee \\
&\qquad (i < k \le n+1 : d'_k = d_{k-1})) \wedge \\
&\quad D^{(x)} = D^{(x)} \cup \vec{d'}) \\
&\quad \vee \\
&\quad (\lceil d_i/s_i \rceil \neq \lfloor d_i/s_i \rfloor \wedge \\
&\quad (\forall d_k : (1 \le k < i : d'_k = d_k \ \wedge d''_k = d_k) \vee \\
&\qquad (k = i : d'_k = s_i \lceil d_i/s_i \rceil \ \wedge \ d''_k = s_i \lfloor d_i/s_i \rfloor) \vee \\
&\qquad (i < k \le n+1 : d'_k = d_{k-1} \ \wedge \ d''_k = d_{k-1})) \wedge \\
&\quad D^{(x)} = D^{(x)} \cup \vec{d'} \cup \vec{d''})).
\end{aligned}
$$

As an example, $D^{(x-1)} = \{(2,-1,0),(0,1,1),(3,3,3)\}$, where loop L_2 is blocked and the blocking factor is three ($s_2 = 3$) results in the following transformed distance vector set $D^{(x)}$:

$$
\begin{aligned}
&\{(2,\lfloor -1/3 \rfloor 3,-1,0),(0,\lfloor 1/3 \rfloor 3,1,1),(3,\lfloor 3/3 \rfloor 3,3,3), \\
&(2,\lceil -1/3 \rceil 3,-1,0),(0,\lceil 1/3 \rceil 3,1,1),(3,\lceil 3/3 \rceil 3,3,3)\}
\end{aligned}
$$

$$
\begin{aligned}
&\{(2,-3,-1,0),(0,0,1,1),(3,3,3,3), \\
&(2,0,-1,0),(0,3,1,1),(3,3,3,3)\}
\end{aligned}
$$

$$
\{(2,-3,-1,0),(0,0,1,1),(3,3,3,3),(2,0,-1,0),(0,3,1,1)\}
$$

As mentioned, all distance vectors before loop blocking have length n, and all distance vectors after blocking a loop have length $n+1$. We need a matrix that, when multiplied by $\vec{d}^{(x-1)} \in D^{(x-1)}$ gives us $\vec{d}^{(x)} \in D^{(x)}$. Therefore, if t_b is the elementary blocking transformation matrix, $\vec{d}^{(x-1)}$ are all $n \times 1$ vectors, and $\vec{d}^{(x)}$ are all $(n+1) \times 1$ vectors, then t_b must be $(n+1) \times n$:

$$
\underbrace{t_b}_{(n+1)\times n} \times \underbrace{\vec{d}^{(x-1)}}_{n\times 1} = \underbrace{\vec{d}^{(x)}}_{(n+1)\times 1}
$$

Blocking may introduce two distance vectors into $D^{(x)}$ where only one existed in $D^{(x-1)}$ because of the ceiling and floor functions. That is, if loops L_i through L_j are each blocked, then each distance vector prior to loop blocking may map into as many as 2^{j-i+1} distance vectors in the transformed distance vector set. Therefore, t_b for mapping distance vectors for a doubly-nested loop is in the form:

$$\text{outer loop} \quad \begin{bmatrix} f(d) & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \begin{bmatrix} g(d) & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\text{inner loop} \quad \begin{bmatrix} 1 & 0 \\ 0 & f(d) \\ 0 & 1 \end{bmatrix}; \quad \begin{bmatrix} 1 & 0 \\ 0 & g(d) \\ 0 & 1 \end{bmatrix}$$

where $f(d) = \lceil d_i/s_i \rceil s_i$ and $g(d) = \lfloor d_i/s_i \rfloor s_i$.

As an example, say the initial distance vector set $D^{(0)}$ for a double nested loop is $\{(2,-1),(0,1),(3,3)\}$ and the transformation matrix that represents reversal of the outer loop and skewing the inner loop prior to loop blocking $T^{(x-1)}$ is:

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

If the outer loop is blocked and $s_1 = 3$, then $T^{(x)} = t_b \times T^{(x-1)}$:

$$\begin{bmatrix} -f(d) & 0 \\ -1 & 0 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} f(d) & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -g(d) & 0 \\ -1 & 0 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} g(d) & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

and the distance vector set after blocking $D^{(x)}$ is $T^{(x)} \times D^{(0)}$:

$$\begin{bmatrix} -f(d) & 0 \\ -1 & 0 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} -f(2) \\ -2 \\ 4-1 \end{bmatrix} = \begin{bmatrix} -\lceil 2/3 \rceil 3 \\ -2 \\ 3 \end{bmatrix} = \begin{bmatrix} -3 \\ -2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} -g(d) & 0 \\ -1 & 0 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} -g(2) \\ -2 \\ 4-1 \end{bmatrix} = \begin{bmatrix} -\lfloor 2/3 \rfloor 3 \\ -2 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 \\ -2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} -f(d) & 0 \\ -1 & 0 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -f(0) \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -\lceil 0/3 \rceil 3 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} -g(d) & 0 \\ -1 & 0 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -g(0) \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -\lfloor 0/3 \rfloor 3 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} -f(d) & 0 \\ -1 & 0 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} -f(3) \\ -3 \\ 6+3 \end{bmatrix} = \begin{bmatrix} -\lceil 3/3 \rceil 3 \\ -3 \\ 9 \end{bmatrix} = \begin{bmatrix} -3 \\ -3 \\ 9 \end{bmatrix}$$

$$\begin{bmatrix} -g(d) & 0 \\ -1 & 0 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} -g(3) \\ -3 \\ 6+3 \end{bmatrix} = \begin{bmatrix} -\lfloor 3/3 \rfloor 3 \\ -3 \\ 9 \end{bmatrix} = \begin{bmatrix} -3 \\ -3 \\ 9 \end{bmatrix}$$

4.1.3 Effect on Source Code Mapping

In the following discussion, the source code shown in Figure 4.3(a) is the initial code. The loop nest is 2-deep; therefore, any matrices representing kernel transformations that occur before loop blocking are $n \times n = 2 \times 2$. The matrix representing blocking of one loop (either L_1 or L_2) is $(n+1) \times n = 3 \times 2$. Any kernel transformations occurring after loop blocking are $(n+1) \times (n+1) = 3 \times 3$.

```
L1   for I1 = 1, m
L2     for I2 = 1, n
S1       A[I1, I2] = ...
```

Figure 4.3(a). Sample code: before transformation.

```
L1'  for J1 = -n, -1
L2'    for J2 = 1, m, s2
L3'      for J3 = max(1, J2), min(m, J2 + s2 - 1)
S1         A[J3, -J1] = ...
```

Figure 4.3(b). Sample code: after transformation.

The transformation matrix representing loop blocking of the inner loop of a doubly-nested loop is:

$$t_b = \begin{matrix} \\ n+1 \end{matrix} \overbrace{\begin{matrix} \\ \end{matrix}}^{n} \left\{ \begin{bmatrix} 1 & 0 \\ 0 & b_o \\ 0 & b_i \end{bmatrix} \right.$$

where b_o is the *blocking* loop and b_i is the *blocked* loop. During composition, $b_i = 1$ and b_o is used to associate the original index variable to the transformed index variable. That is:

$$t_b = \begin{bmatrix} 1 & 0 \\ 0 & b_o \\ 0 & 1 \end{bmatrix}$$

As an example, say the following sequence of transformations is applied to the source code shown in Figure 4.3(a):

1. Permute both loops (**permute**(L_1, L_2));
2. Block the inner loop by a factor of s_2 (**block**(L_2, s_2));
3. Reverse the outermost loop (**reverse**(L_1)).

then the matrix representing the sequence of transformations is:

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

Mapping the assignment statements follows the source code mapping algorithm. The initial loop nest is 2 deep ($n = 2$). After blocking, the source code is 3 deep and the new index variables are J_1, J_2, and J_3. The transformed index variables are calculated in terms of the initial index variables using the transformation matrix T representing loop permutation, blocking, and reversal.

$$\begin{bmatrix} J_1 \\ J_2 \\ J_3 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ b_o & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} \qquad \begin{array}{ll} J_1 = -I_2; & I_2 = -J_1; \\ J_2 = b_o(I_1); & \\ J_3 = I_1; & I_1 = J_3 \end{array}$$

That is, in the assignment statements, I_2 is replaced with $-J_1$ and I_1 is replaced with J_3. The assignment statement S_1 in Figure 4.3(a) is transformed as follows:

$$\mathtt{A}[I_1, I_2] \Rightarrow \mathtt{A}[J_3, -J_1]$$

Next the loop bounds are transformed. The inequalities, minima, and maxima are determined as in the original source code mapping algorithm.

Extract Inequalities	Find min's and max's
$I_1 \geq 1; \quad I_1 \leq m;$	$\mathbf{min}(I_1) = 1; \quad \mathbf{max}(I_1) = m;$
$I_2 \geq 1; \quad I_2 \leq n$	$\mathbf{min}(I_2) = 1; \quad \mathbf{max}(I_2) = n$

When the indices are transformed, four new inequalities are introduced into the inequality set to reflect the new blocking and blocked index variables. Specifically, if loop L_i with index variable I_i is blocked by a blocking factor of s_i and becomes a blocking loop (b_o) with index variable J_s and a blocked loop (b_i) with index variable J_t, then the following inequalities are added to the inequality set:

$$\boxed{\begin{array}{l} J_s \geq lb_i; \quad J_s \leq ub_i; \\ J_t \geq J_s; \quad J_t \leq J_s + s_i - 1. \end{array}}$$

Using the source code shown in Figure 4.3(a), the inequality set becomes:

$$\begin{array}{llll} J_3 \geq 1; & J_3 \leq m; & \mathbf{min}(J_3) = 1; & \mathbf{max}(J_3) = m; \\ -J_1 \geq 1; & -J_1 \leq n; & \mathbf{min}(-J_1) = 1; & \mathbf{max}(-J_1) = n; \\ J_2 \geq 1; & J_2 \leq m; & & \\ J_3 \geq J_2; & J_3 \leq J_2 + s_2 - 1. & & \end{array}$$

The final step, calculating the new loop bounds, is equivalent for blocked code and code containing only kernel transformations. That is, the entire inequality set is used to determine the new loop bounds for J_1, J_2, and J_3, where the boxed inequalities are used to determine the lower bounds and upper bounds of J_1, J_2, and J_3.

J_1		J_2		J_3	
$-J_1 \geq 1$;	$-J_1 \leq n$;	$\boxed{J_2 \geq 1}$	$\boxed{J_2 \leq m}$	$\boxed{J_3 \geq 1}$	$\boxed{J_3 \leq m}$
$\boxed{J_1 \leq -1}$	$\boxed{J_1 \geq -n}$			$\boxed{J_3 \geq J_2}$	$\boxed{J_3 \leq J_2 + s_2 - 1}$
		$J_2 \leq J_3$;	$J_2 \geq J_3 - s_2 + 1$;		
		$J_2 \leq \max(J_3)$;	$J_2 \geq \min(J_3) - s_2 + 1$;		
		$\boxed{J_2 \leq m}$	$J_2 \geq 1 + 1 - s_2$;		
			$\boxed{J_2 \geq 2 - s_2}$		

Given the bounds calculated above, the transformed source code is shown in Figure 4.3(b).

4.2 Loop Collapsing

Loop collapsing is the inverse of loop blocking and decreases an n-nested loop to an $n-1$ nested loop [14, 15, 18]. Repeated application of loop collapsing is called *loop coalescing* and results in an n'-nested loop, where $n' < n$. The source code shown in Figure 4.4(a) is transformed into the source code shown in Figure 4.4(b) by loop collapsing. Additional statements are introduced to recalculate the value of the index variables I_1 and I_2 in the transformed source code (S_i').

A loop nest may be collapsed to improve scheduling, reduce barrier synchronization, or improve utilization of the memory hierarchy. Scheduling is improved because the work is divided among fewer loops, and self-scheduling techniques are more apt to find work available when the size of the task varies. Barrier synchronization is reduced because fewer synchronization points exist due to the decrease in the number of loop nests. Finally, loop collapsing increases the effective vector length for vector machines, improving utilization of vector registers. Vector operations are typically applied to the loop with the highest iteration count. Collapsing two adjacent loops L_i and L_j results in a collapsed loop L_k with a greater iteration count than loops L_i or L_j had individually.

```
      integer W[2,3], V[2,3]

L1    for I1 = 1, ub_i
L2     for I2 = 1, ub_j
S1      W[I1, I2] = V[I1, I2] * 5
```

Figure 4.4(a). Sample code: before *loop collapsing* transformation.

```
      integer W[2,3], V[2,3]

L1,2  for I1,2 = 1, ub_i × ub_j
S'1     if ((I1,2 mod ub_j) == 0)
S'2      then I1 = I1,2/ub_j
S'3      else I1 = ceil(I1,2/ub_j) + 1
S'4     I2 = ((I1,2 − 1) mod ub_j) + 1
S1      W[I1, I2] = V[I1, I2] * 5
```

Figure 4.4(b). Sample code: after *loop collapsing* transformation.

4.2.1 Legality Criteria

The legality criteria for loop collapsing was initially described by Wolfe [14], and later formally specified by Polychronopoulos [27]. However, this legality condition is too restrictive because it results in the illegality of loop collapsing in many cases where its application results in semantically equivalent source code. Wolfe and Polychronopoulos state that two loops L_i and L_j can be collapsed as long as d_i and d_j are 0 for all of the direction vectors in the direction vector set. According to Wolfe and Polychronopoulos, loops L_i and L_j can be collapsed if:

$$(\forall \vec{d}^{(x-1)} : \vec{d}^{(x-1)} \in D^{(x-1)} : d_i = 0 \wedge d_j = 0).$$

As a counter-example to this legality criteria, the source code in Figure 4.5(a) can be collapsed into the source code in Figure 4.5(b), resulting in semantically equivalent source code, where $D = \{(1,2)\}$ $(d_1 \neq 0 \wedge d_2 \neq 0)$.

In this dissertation, it is legal to collapse two loops, L_i and L_j, as long as loop L_i is the parent loop L_j in the loop nest hierarchy and loops L_i and L_j are both normal. Loop collapsing cannot be applied to two loops L_i and L_j if they are imperfectly nested. Collapsing code that is imperfectly nested results in an increased number of iterations for the statement S_k between the loops, which results in S_k executing more times than in the initial source code. As an example, statement S_1 executes m times in the source code shown

```
L1   for I1 = 1,2
L2    for I2 = 1,3
S1     W[I1,I2] = ...
S2           ... = W[I1 - 1, I2 - 2]
```

Figure 4.5(a). Sample code: before *loop collapsing* transformation, illegal by Wolfe's [14] and Polychronopoulos's [27] definition.

```
L1,2  for I1,2 = 1,6
S'1    if ((I1,2 mod 3) == 0)
S'2     then I1 = I1,2/3
S'3     else I1 = ceil(I1,2/3) + 1
S'4    I2 = ((I1,2 - 1) mod 3) + 1
S1     W[I1,I2] = ...
S2           ... = W[I1 - 1, I2 - 2]
```

Figure 4.5(b). Sample code: after legal application of *loop collapsing* transformation.

in Figure 4.6(a). However, statement S_1 executes $m \times n$ times in the collapsed source code shown in Figure 4.6(b).

```
L1   for I1 = 1,m
S1    A[I1] = ...
L2    for I2 = 1,n
```

Figure 4.6(a). Sample code: imperfectly nested code before *loop collapsing* transformation.

```
L1,2  for I1,2 = 1, m × n
       if((I1,2 mod n) == 0)
        then I1 = I1,2/n
        else I1 = ceil(I1,2/n) + 1
       I2 = ((I1,2 - 1) mod n) + 1
S1     A[I1] = ...
```

Figure 4.6(b). Sample code: illegal *loop collapsing* transformation of imperfectly nested code.

In summary, the legality for loop collapsing is:

Preconditions

Invariant

$$\boxed{normal(B^{(x-1)}, i) \wedge normal(B^{(x-1)}, j)}$$

Explicit

$$\boxed{j = i + 1}$$

Nesting

$$\boxed{perfect(L_i, L_j)}$$

4.2.2 Effect on the Dependence Vector Set D

The initial loop nest depth is n, and all dependence vectors prior to loop collapsing $\vec{d}^{(x-1)}$ in $D^{(x-1)}$ have a length n, $(\forall \vec{d}^{(x-1)} : \vec{d}^{(x-1)} \in D^{(x-1)} : |\vec{d}^{(x-1)}| = n)$. The dependence vector set after loop collapsing is $D^{(x)}$, if loop collapsing is the x^{th} transformation. As mentioned earlier, the loop nest depth decreases by one in loop collapsing and correspondingly, the length of the dependence vectors $\vec{d}^{(x)}$ decreases by one, $(\forall \vec{d}^{(x)} : \vec{d}^{(x)} \in D^{(x)} : |\vec{d}^{(x-1)}| = n-1)$.

Each dependence vector $\vec{d}^{(x)}$ in the dependence vector set $D^{(x)}$ is modified as follows:

- distances in loops outside of the collapsed code $(1 \leq k < i)$ remain unchanged;
- distances within the collapsed loops $(k = i)$ are assigned a value that is a function of the initial distances and the upper bound of the inner loop;
- distances in loops inside of the collapsed code $(j \leq k \leq n-1)$ are assigned the original distances with locations offset by one.

Formally, the modification to the distances in each distance vector when loops L_i and L_j are collapsed is specified as follows, where ub_j is the upper bound for loop L_j:

$$
\begin{aligned}
(\forall \vec{d}^{(x-1)} : \ & \vec{d}^{(x-1)} \in D^{(x-1)} : \\
& ((\forall k : 1 \leq k < i : d'_k = d_k) \vee \\
& (k = i : d'_k = (d_k \times ub_j) + d_{k+1}) \vee \\
& (\forall k : j \leq k \leq n-1 : d'_k = d_{k+1})) \wedge \\
& D^{(x)} = D^{(x)} \cup \vec{d'})
\end{aligned}
$$

As an example, consider a distance vector set before loop collapsing $D^{(x-1)} = \{(0,2,-1,3),(0,1,1,2)\}$. Loop collapsing is applied to loops L_2 and L_3 and the upper bound of loop L_3 is 2 ($lb_3 = 2$). The resulting distance vector set $D^{(x)} = \{(0,(2 \times ub_3) - 1,3),(0,(1 \times ub_3) + 1,2)\}$, which is $\{(0,3,3),(0,3,2)\}$.

As with loop blocking, the transformation matrix representing the effect of loop collapsing is represented by an elementary matrix t_c. If loop collapsing is the x^{th} transformation, then all $\vec{d}^{(x-1)} \in D^{(x-1)}$ have length n and all $\vec{d}^{(x)} \in D^{(x)}$ have length $n-1$. Therefore, the

matrix t_c is $(n-1) \times n$:

$$\underbrace{t_c}_{(n-1)\times n} \times \underbrace{\vec{d}^{(x-1)}}_{n\times 1} = \underbrace{\vec{d}^{(x)}}_{(n-1)\times 1}$$

Suppose that a triply-nested loop is collapsed, then the transformation matrix t_c that represents collapsing of the outermost loops, L_1 and L_2, is in the form:

$$t_c = \begin{bmatrix} ub_2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and the transformation matrix t_c that represents loop collapsing of the innermost loops, L_2 and L_3, is in the form:

$$t_c = \begin{bmatrix} 1 & 0 & 0 \\ 0 & ub_3 & 1 \end{bmatrix}$$

As an example, if $D^{(x-1)} = \{(2,-1,3),(0,1,1),(5,6,-7)\}$, loops L_2 and L_3 are collapsed, and the upper bound of the inner loop is two ($ub_3 = 2$), then $D^{(x)}$ is calculated as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & ub_3 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2(-1)+1(3) \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & ub_3 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2(1)+1(1) \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & ub_3 & 1 \end{bmatrix} \times \begin{bmatrix} 5 \\ 6 \\ -7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 5 \\ 61 \\ -7 \end{bmatrix} = \begin{bmatrix} 5 \\ 2(6)+1(-7) \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$$

4.2.3 Effect on Source Code Mapping

In the following discussion, the triply-nested loop shown in Figure 4.7(a) is the initial code. Loop collapsing maps the triply-nested loop ($n = 3$) into a doubly-nested loop ($n = 2$). Any kernel transformations occurring before collapsing are represented by an $n \times n$ matrix. The matrix representing loop collapsing t_c is $(n-1) \times n$, and any transformations occurring after loop collapsing are represented by $(n-1) \times (n-1)$ matrices.

```
L1   for I1 = 1, ub1
L2    for I2 = 1, ub2
L3     for I3 = 1, ub3
S1     A[I1, I2, I3] = ...
```

Figure 4.7(a). Sample code: before transformation.

```
L1'  for J1 = ub2
L2'   for J2 = 1, ub1 × ub3
       if(J2 mod ub3) == 0
        then I1 = J2/ub3
        else I1 = ⌈J2/ub3⌉ + 1
       I2 = ((J2 − 1) mod ub3) + 1
S1     A[I1, J1, I3] = ...
```

Figure 4.7(b). Sample code: after transformation.

The transformation matrix that represents the collapsing of the inner two loops of a triply-nested loop is:

$$t_c = n-1 \left\{ \overbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & c_o & c_i \end{bmatrix}}^{n} \right.$$

where c_o is the outer loop and c_i is the inner loop of the two loops that are collapsed. During composition, the initial index variables associated with c_o and c_i are significant to the source code mapping.

As an example, the following sequence of transformations is applied to the source code shown in Figure 4.7(a):

1. Permute the outer two loops (**permute**(L_1, L_2));
2. Collapse the inner two loops (**collapse**(L_2, L_3));

The transformation matrix that represents the sequence is:

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

As with loop blocking, source code mapping follows the mapping algorithm with some minor modifications. The assignment statements are mapped from the initial index variables I_1, I_2, and I_3 to the transformed index variables J_1 and J_2.

$$\begin{bmatrix} J_1 \\ J_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ c_o & 0 & c_i \end{bmatrix} \times \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} \qquad \begin{array}{l} J_1 = I_2 \\ J_2 = c_o(I_1), c_i(I_3) \end{array}$$

The mapping shows that I_2 is replaced with J_1 in the transformed source code. Also, the outer collapsed loop (c_o) is I_1 and the inner collapsed loop (c_i) is I_3.

$$\mathtt{A}[I_1, I_2, I_3] \Rightarrow \mathtt{A}[I_1, J_1, I_3]$$

Note that the index variables I_1 and I_3 are not transformed in the final source code. As will be shown later, the indices are recalculated by some additional assignment statements.

The loops are mapped after the index variables are calculated. The inequalities, minima, and maxima are determined from the initial source code.

Extract Inequalities	Find min's and max's
$I_1 \geq 1; \quad I_1 \leq ub_1;$	$\mathbf{min}(I_1) = 1; \quad \mathbf{max}(I_1) = ub_1;$
$I_2 \geq 1; \quad I_2 \leq ub_2;$	$\mathbf{min}(I_2) = 1; \quad \mathbf{max}(I_2) = ub_2;$
$I_3 \geq 1; \quad I_3 \leq ub_3$	$\mathbf{min}(I_3) = 1; \quad \mathbf{max}(I_3) = ub_3$

The indices are transformed using inequalities from the initial source code and the index variable mapping used for the assignment statements. Two inequalities for the index variables that represent the collapsed loop replace the initial inequalities. That is, if loops L_i and L_j are collapsed and J_s is the index variable in the transformed source code for the collapsed loops, then the following inequalities are introduced:

$$J_s \geq lb_i \times lb_j; \qquad J_s \leq ub_i \times ub_j$$

If we assume that the loops are normalized prior to loop collapsing ($lb_i = 1$ and $lb_j = 1$), then the first inequality is replaced with $J_s \geq 1 \times 1 \equiv J_s \geq 1$.

Continuing with the example, the transformed inequality set is:

J_1	J_2
$\boxed{J_1 \geq 1}$ $\boxed{J_1 \leq ub_2}$	$J_2 \geq lb_i \times lb_j$; $\boxed{J_2 \leq ub_i \times ub_j}$ $J_2 \geq 1 \times 1$; $\boxed{J_2 \geq 1}$

Also, after loop collapsing, the array subscripts for I_i and I_j are recalculated. The recalculated array subscripts for I_i and I_j with J_s representing the collapsed loops are:

```
if ((J_s mod ub_j) == 0)
    then I_i = J_s/ub_j
    else I_i = ceil(J_s/ub_j) + 1
I_j = ((J_s − 1)mod ub_j) + 1
```

The complete transformed source code is shown in Figure 4.7(b).

CHAPTER 5

Loop Fission and Loop Fusion

Loop fission is a transformation that changes the structure of a loop nest, such that it becomes two adjacent loop nests. *Loop fusion* is the inverse of loop fission and combines two adjacent loop nests into one. The following discussion describes the legality criteria, effect on the distance vector set D, and effect on the transformation matrix T for both loop fission and loop fusion [12, 22].

5.1 Loop Fission

Loop Fission [17, 18, 19, 20, 21] is a loop transformation technique that divides a set of program statements that are within a nested loop into two adjacent nested loops. As an example, the source code in Figure 5.1(a) becomes the source code in Figure 5.1(b) after loop fission. Applying loop fission improves the localities of reference such that the resultant code has the largest possible groupings of references to the same variables, and also may enable unimodular transformations, either by modifying the transformed source code so that all assignment statements are perfectly nested [28], or by reducing the cardinality of the distance vector set D.

5.1.1 Legality Criteria

The initial source code has a dependence vector set $D^{(0)}$ that contains all dependence vectors between statements in the initial code. As mentioned earlier, $D^{(0)}$ can be represented by

```
L1   for I1 = 1,n
L2    for I2 = 1,m
L3     for I3 = 1,p
S1      A[I1,I2,I3] = B[I1 - 1,I2,I3]
S2      C[I1,I2,I3] = D[I1,I2 - 3,I3]
S3      B[I1,I2,I3] = A[I1,I2 - 2,I3 - 1]
S4      D[I1,I2,I3] = B[I1 + 1,I2 - 1,I3]+
                      C[I1,I2,I3 - 2]
S5      var1 = var2
```

Figure 5.1(a). Sample code: before *loop fission* transformation

```
L1   for I1 = 1,n
L2    for I2 = 1,m
L3     for I3 = 1,p
S2      C[I1,I2,I3] = D[I1,I2 - 3,I3]
S4      D[I1,I2,I3] = B[I1 + 1,I2 - 1,I3]+
                      C[I1,I2,I3 - 2]
L1   for I1 = 1,n
L2    for I2 = 1,m
L3     for I3 = 1,p
S1      A[I1,I2,I3] = B[I1 - 1,I2,I3]
S3      B[I1,I2,I3] = A[I1,I2 - 2,I3 - 1]
S5      var1 = var2
```

Figure 5.1(b). Sample code: after *loop fission* transformation

a dependence graph, where each assignment statement is a node and each dependence is a directed arc. A Π-block is a strongly connected component of the dependence graph [13]. Π-blocks are significant because they cannot be divided by most loop transformations. A Ψ-block (defined in this paper) is a set of statements whose members are strongly connected components. In other words, dividing a set of statements into two Ψ-blocks (Ψ_1 and Ψ_2) results in two partitions such that each element of Ψ_1 and Ψ_2 is a Π-block.

The source code in Figure 5.1(a) has three Π-blocks: Π_1 = (S1,S3), Π_2 = (S2,S4), and Π_3 = (S5). After transformation, the source code shown in Figure 5.1(b) has two Ψ-blocks: Ψ_1 = (S2,S4), and Ψ_2 = (S1,S3,S5). Note that none of the Π-blocks are divided by the loop fission transformation.

In the following discussion, loop fission is the x^{th} transformation in the sequence. The dependence vector set for the statements in Ψ_1 is $D^{(x-1)}_{\Psi_1,\Psi_1}$, and the dependence vector set for the statements in Ψ_2 is $D^{(x-1)}_{\Psi_2,\Psi_2}$. Finally, the dependence vector set *from* Ψ_1 *to* Ψ_2 is labeled $D^{(x-1)}_{\Psi_1,\Psi_2}$ and the dependence vector set *from* Ψ_2 *to* Ψ_1 is $D^{(x-1)}_{\Psi_2,\Psi_1}$. The union of all four of these sets is $D^{(x-1)}$, the dependence vector set before the loop fission transformation:

$$D^{(x-1)} = D^{(x-1)}_{\Psi_1,\Psi_1} \cup D^{(x-1)}_{\Psi_2,\Psi_2} \cup D^{(x-1)}_{\Psi_1,\Psi_2} \cup D^{(x-1)}_{\Psi_2,\Psi_1}.$$

As an example, the dependence vector set $D^{(x-1)}$ for the source code in Figure 5.1(a) is:

$$D^{(x-1)} = \{(0,2,1),(0,0,2),(1,0,0),(1,-1,0),(0,3,0)\}, \tag{5.1}$$

where $D^{(x-1)}_{\Psi_2,\Psi_2} = \{(0,2,1),(1,0,0)\}$, $D^{(x-1)}_{\Psi_1,\Psi_1} = \{(0,0,2),(0,3,0)\}$, $D^{(x-1)}_{\Psi_2,\Psi_1} = \emptyset$, and $D^{(x-1)}_{\Psi_1,\Psi_2} = \{(1,-1,0)\}$. As can be seen from Expression (5.1), $D^{(x-1)} = D^{(x-1)}_{\Psi_1,\Psi_1} \cup D^{(x-1)}_{\Psi_2,\Psi_2} \cup D^{(x-1)}_{\Psi_1,\Psi_2} \cup D^{(x-1)}_{\Psi_2,\Psi_1}$.

The dependence vector sets $D^{(x-1)}_{\Psi_1,\Psi_2}$ and $D^{(x-1)}_{\Psi_2,\Psi_1}$ are critical to the legality and order of the loop fission transformation. If there are dependences from both Ψ_1 to Ψ_2 and from Ψ_2 to Ψ_1, then loop fission cannot be applied. A non-empty $D^{(x-1)}_{\Psi_1,\Psi_2}$ implies that Ψ_1 must execute before Ψ_2 in the transformed source code. Similarly, a non-empty $D^{(x-1)}_{\Psi_2,\Psi_1}$ implies that Ψ_2 must execute before Ψ_1. Clearly, both execution orders cannot be satisfied.

Formally, loop fission *cannot* be applied if $((\exists \vec{d} : \vec{d} \in D^{(x-1)}_{\Psi_1,\Psi_2}) \wedge (\exists \vec{d} : \vec{d} \in D^{(x-1)}_{\Psi_2,\Psi_1}))$. A simpler, formal notation to describe the legality criteria is:

Preconditions

Invariant

$$\boxed{\text{TRUE}}$$

Explicit

$$\boxed{(D^{(x-1)}_{\Psi_1,\Psi_2} = \emptyset \vee D^{(x-1)}_{\Psi_2,\Psi_1} = \emptyset)}$$

Nesting

$$\boxed{\text{TRUE}}$$

This notation states that loop fission is legal if either the distance vector set from Ψ_1 to Ψ_2, or from Ψ_2 to Ψ_1, is empty, regardless of the nesting structure of the two Ψ-blocks.

5.1.2 Effect on the Dependence Vector Set D

The dependence vector set for the source code before the loop fission transformation is $D^{(x-1)} = D^{(x-1)}_{\Psi_1,\Psi_1} \cup D^{(x-1)}_{\Psi_2,\Psi_2} \cup D^{(x-1)}_{\Psi_1,\Psi_2} \cup D^{(x-1)}_{\Psi_2,\Psi_1}$. The sets $D^{(x-1)}_{\Psi_1,\Psi_2}$ and $D^{(x-1)}_{\Psi_2,\Psi_1}$ were initially loop-carried dependences; however, after loop fission the partitions $D^{(x-1)}_{\Psi_1,\Psi_2}$ and $D^{(x-1)}_{\Psi_2,\Psi_1}$ become loop-independent. Loop fission is referred to as a dependence-breaking transformation [17]

because the dependences *between* Ψ-blocks are carried at a higher level after transformation. After loop fission, the dependence vector set for the first code block ($D^{(x)}_{\Psi_1,\Psi_1}$) consists of the dependences within the statement partition Ψ_1. Similarly, the dependence vector set for the second code block ($D^{(x)}_{\Psi_2,\Psi_2}$) consists of the dependences within the statement partition Ψ_2:

Partition Ψ_1:	$D^{(x)} = (D^{(x-1)}_{\Psi_1,\Psi_1})$
Partition Ψ_2:	$D^{(x)} = (D^{(x-1)}_{\Psi_2,\Psi_2})$

5.1.3 Effect on Source Code Mapping

Any unimodular transformation can be applied at any time during the loop transformation process (composition property). In compliance with this property, loop fission can be applied at any point in the parallelizing process, given that the legality criteria are met.

Loop fission results in two adjacent nested loops where only one previously existed. Therefore, both new blocks of code (Ψ_1 and Ψ_2) have a transformation matrix associated with them ($T^{(x)}_{\Psi_1}$ and $T^{(x)}_{\Psi_2}$). Because of the composition property, both $T^{(x)}_{\Psi_1}$ and $T^{(x)}_{\Psi_2}$ are initialized to the transformation matrix that existed for the nested loop prior to loop fission ($T^{(x-1)}$). Notationally, this condition is expressed as:

Partition Ψ_1:	$T^{(x)}_{\Psi_1} = T^{(x-1)}$
Partition Ψ_2:	$T^{(x)}_{\Psi_2} = T^{(x-1)}$

5.2 Loop Fusion

This section describes the extension of the unimodular approach to include *loop fusion* [17, 18, 19], which is the complement of *loop fission*. Loop fusion combines two adjacent nested loops into one nested loop resulting in a decrease in communication and synchronization overhead. As an example, the initial source code that is shown in Figure 5.2(a) is transformed into the source code shown in Figure 5.2(b) by loop fusion.

```
L1   for I1 = 1,n
L2    for I2 = 1,m
L3     for I3 = 1,p
S1      A[I1, I2, I3] = B[I1, I2, I3]
S2      A[I1 - 1, I2, I3 - 2] = D[I1, I2 - 2, I3]
L1'  for I1 = 1,n
L2'   for I2 = 1,m
L3'    for I3 = 1,p
S3      E[I1 - 2, I2, I3] = F[I1, I2, I3] + G[I1, I2, I3]
S4      H[I1 - 3, I2, I3] = E[I1, I2, I3]
```

Figure 5.2(a). Sample code: before *loop fusion* transformation

```
L1   for I1 = 1,n
L2    for I2 = 1,m
L3     for I3 = 1,p
S1      A[I1, I2, I3] = B[I1, I2, I3]
S2      A[I1 - 1, I2, I3 - 2] = D[I1, I2 - 2, I3]
S3      E[I1 - 2, I2, I3] = F[I1, I2, I3] + G[I1, I2, I3]
S4      H[I1 - 3, I2, I3] = E[I1, I2, I3]
```

Figure 5.2(b). Sample code: after *loop fusion* transformation

5.2.1 Legality Criteria

Previous work related to loop fusion implies that the complexity of determining the legality of the transformation exceeds the potential gains from its implementation [14]. This section of the paper clarifies and simplifies the legality condition for loop fusion based upon properties of the loop bound set and the distance vector set.

A Θ-*block* is a set of statements that is associated by spatial locality in the source code. That is, all of the assignment statements in a loop nest make up one Θ-block. As an example, the source code in Figure 5.2(a) has two Θ-blocks: Θ_1 = (S1,S2) and Θ_2 = (S3,S4). A Θ-block is another name for a loop nest and is introduced in this paper to ensure consistent notation with loop fission. Unlike Π- and Ψ-blocks, nothing is assumed about the dependence relations within Θ-blocks .

Zima [15] and Wolfe [14] both state that a precondition for loop fusion is that the loop bound sets for both loop nests must match exactly, or be "adjustable" using a transformation such as *loop unrolling.* The following discussion shows that this precondition can be weakened into a predicate named `can_adjust`($B_{\Theta_1}, B_{\Theta_2}$), which returns *true* if the loop bound sets for Θ_1 (B_{Θ_1}) and Θ_2 (B_{Θ_2}) can be adjusted to match exactly. To clarify, Zima and Wolfe state that the loop bound sets must *already* match, whereas `can_adjust`() de-

termines if the loop bound sets *can be* adjusted. An algorithm that determines the value of **can_adjust**($B_{\Theta_1}, B_{\Theta_2}$) is given in Figure 5.3.

Algorithm can_adjust
Given two loop bound sets, B_{Θ_1} and B_{Θ_2}, determines if the loop bound sets can be adjusted. Evaluation of this function is a precondition for loop fusion.

Input:

B_{Θ_1} /***loop bound set for the first code block***/
B_{Θ_2} /***loop bound set for the second code block***/

Output:

FLAG /***TRUE or FALSE***/

Procedure:

1. /***Determine iteration count for all 3-tuples in* B_{Θ_1} *and* B_{Θ_2}**/
   For all $\vec{b_i} \in B_{\Theta_1}$
   $\Theta_1\_count_i = \lfloor (lb_i - ub_i + s_i)/s_i \rfloor$

   For all $\vec{b_j} \in B_{\Theta_2}$
   $\Theta_2\_count_j = \lfloor (lb_j - ub_j + s_j)/s_j \rfloor$

2. /***Sort* Θ_1_count *and* Θ_2_count**/
   SORT(Θ_1_count)
   SORT(Θ_2_count)

3. /***Determine value of FLAG***/
   FLAG = TRUE
   For all $i \in \Theta_1\_count$
   /***if the iteration counts differ for a loop in* Θ_1 *and* Θ_2**/
   If $\Theta_1\_count_i \neq \Theta_2\_count_i$
   /***then the two loop bound sets cannot be adjusted***/
   Then FLAG = FALSE

Figure 5.3. Algorithm to determine if two loop bound sets can be adjusted.

Determining the set of transformations needed to adjust B_{Θ_1} and B_{Θ_2} may involve a complex solution. Note that **can_adjust**($B_{\Theta_1}, B_{\Theta_2}$) determines *if* the loop bound sets can be adjusted, but does not determine the actual adjustments. The algorithm to determine the adjustment matrix is dependent upon the cardinality of the set of transformations and the loop nest depth of the target source code.

Typically, another legality criteria for loop fusion is based upon the dependences between the two code blocks Θ_1 and Θ_2. The dependence vector set after loop fusion has four partitions:

1. $D^{(x-1)}_{\Theta_1,\Theta_1}$: The loop-carried dependences between statements in Θ_1.
2. $D^{(x-1)}_{\Theta_2,\Theta_2}$: The loop-carried dependences between statements in Θ_2.
3. $D^{(x)}_{\Theta_1,\Theta_2}$: Before the fusion transformation, this partition was loop-independent. However, after loop fusion, this partition is loop-carried and is the distance vector set *from* statements in Θ_1 *to* statements in Θ_2.
4. $D^{(x)}_{\Theta_2,\Theta_1}$: Before the fusion transformation, this partition was loop-independent. However, after loop fusion, this partition is loop-carried and is the distance vector set *from* statements in Θ_2 *to* statements in Θ_1.

As an example, the dependence vector sets for the initial source code in Figure 5.4(a) are: $D^{(x-1)}_{\Theta_1,\Theta_1} = \{(0)\}$ and $D^{(x-1)}_{\Theta_2,\Theta_2} = \{(0)\}$, where Θ_1 =(S1) and Θ_2 =(S2). Note that there is also a loop-independent dependence from S2 to S1 caused by variable A[i]. After loop fusion is applied, as shown in Figure 5.4(b), the loop-independent dependences from S2 to S1 are now loop-carried ($D^{(x)}_{\Theta_1,\Theta_2} = \emptyset$, and $D^{(x)}_{\Theta_2,\Theta_1} = \{(1)\}$).

Wolfe [14] describes the legality of loop fusion based upon dependences between statements in the first loop nest Θ_1 and the second loop nest Θ_2. Zima [15] uses a similar, although more specific definition, called *serial-fusion preventing*, to describe an illegal condition for applying loop fusion. Both discussions state that loop fusion is illegal if $D^{(x)}_{\Theta_2,\Theta_1} \neq \emptyset$. In other words, if after the loop fusion transformation there exists any loop-carried dependences from the second to the first code block, then loop fusion is illegal. According to this legality criteria, the source code shown in Figure 5.4(a) cannot be fused and still maintain semantic equivalence to the original source code. For illustrative purposes, the source code in Figure 5.4(b) shows the fused code, and, as can be seen, it is not semantically equivalent to the initial code because of the dependence from S_2 to S_1 caused by array A. However, if the loop is reversed after loop fusion is applied, as shown in Figure 5.4(c), then the transformed source code is semantically equivalent to the initial code.

This example contradicts the legality criteria proposed by Wolf and Zima. That is, we are able to apply loop fusion to code that initially appeared non-fusible. From studying and

exploiting properties of the dependence vector set we developed a new strategy for applying loop fusion. The approach for determining the legal application of loop fusion is as follows.

```
L1   for I1 = 1,n
S1     A[I1] = A[I1] + C[I1]
L1'  for I1 = 1,n
S2     D[I1] = A[I1 + 1] + D[I1]
```

Figure 5.4(a). Sample code: before *loop fusion* transformation.

```
L1   for I1 = 1,n
S1     A[I1] = A[I1] + C[I1]
S2     D[I1] = A[I1 + 1] + D[I1]
```

Figure 5.4(b). Sample code: *loop fusion* results in non-equivalent source code.

```
L1   for I1 = -n,-1
S1     A[-I1] = A[-I1] + C[-I1]
S2     D[-I1] = A[-I1 + 1] + D[-I1]
```

Figure 5.4(c). Sample code: *loop fusion* followed by *loop reversal* results in equivalent source code.

It is understood that distance vectors are initially lexicographically positive and define the lexicographic execution order of the assignment statements. In the initial source code (Figure 5.4(a)), all iterations of statement S_1 complete execution before any iterations of statement S_2 begin execution. Clearly, the source code after loop fusion must preserve any dependences between S_1 and S_2.

The distance vectors in $D^{(x)}_{\Theta_2,\Theta_1}$ must reflect the execution order of Θ_1 before Θ_2. Therefore, the set $D^{(x)}_{\Theta_2,\Theta_1}$ is negated when $D^{(x)}$ is determined after the loop fusion transformation. That is, the distance vector set $D^{(x)}$ is:

$$D^{(x)} = D^{(x-1)}_{\Theta_1,\Theta_1} \cup D^{(x-1)}_{\Theta_2,\Theta_2} \cup D^{(x)}_{\Theta_1,\Theta_2} \cup -(D^{(x)}_{\Theta_2,\Theta_1}).$$

As an example, the distance vector set for the source code in Figure 5.4(b) is:

$$D^{(x)} = \{(0)\} \cup \{(0)\} \cup \emptyset \cup -\{(1)\} = \{(0),(-1)\}$$

Determining $D^{(x)}$ after loop fusion may result in lexicographically non-positive dependence vectors in $D^{(x)}_{\Theta_2,\Theta_1}$, as in the above case when $D^{(x)}_{\Theta_2,\Theta_1} = -\{(1)\}$. Even though some distance vectors may not be lexicographically positive during intermediate transformations, the global postcondition must be *true* when all transformations are complete. That is, all

of the final, transformed distance vectors must be lexicographically positive. The reversal transformation is applied after loop fusion in the source code in Figure 5.4(c), which makes all of the distance vectors in $D^{(x)}$ lexicographically positive.

A less trivial example is illustrated by the source code in Figure 5.5(a), which has only one distance vector ($D^{(x)}_{\Theta_2,\Theta_1} = \{(1,-1)\}$) after loop fusion is applied ($D^{(x-1)}_{\Theta_1,\Theta_1}, D^{(x-1)}_{\Theta_2,\Theta_2}$, and $D^{(x)}_{\Theta_1,\Theta_2}$ are all empty). Reversal of the signs of the distances in $D^{(x)}_{\Theta_2,\Theta_1}$ results in a distance vector set that reflects the execution order of Θ_1 before Θ_2 ($D^{(x)} = -(D^{(x)}_{\Theta_2,\Theta_1}) = -\{(1,-1)\} = \{(-1,1)\}$).

The distance vector in $D^{(x)}$ is not lexicographically positive, and therefore the global postcondition is *false*. Two possible transformations that make the distance vector set lexicographically positive are to: 1) reverse the outermost loop using the transformation matrix:

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

or 2) interchange the inner and outer loop using the transformation matrix:

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

Loop reversal applied after loop fusion is shown in Figure 5.5(b) and loop interchange applied after loop fusion is shown in Figure 5.5(c). Both cases produce semantically equivalent code with respect to the initial source code. Loop reversal results in $D^{(x)} = \{(1,1)\}$, and the transformed source code is semantically equivalent to the initial source code because the transformed distance vector set is lexicographically positive. Loop permutation results in $D^{(x)} = \{(1,-1)\}$, and, again, the distance vector set is lexicographically positive and the transformation sequence is legal.

The above discussion shows that the legality criteria for loop fusion need not be based on the dependence vector set between the two fused blocks of code. Instead, the legality criteria for loop fusion is based strictly upon the predicate **can_adjust**($B_{\Theta_1}, B_{\Theta_2}$). As with all unimodular transformation sequences, all $\vec{d}$ in $D^{(x)}$ must be lexicographically positive

```
L1   for I1 = 1,3
L2    for I2 = 1,3
S1     A[I1,I2] = ...
L1'  for I1 = 1,3
L2'   for I2 = 1,3
S2     ... = A[I1 + 1,I2 - 1]
```

Figure 5.5(a). Sample code: before *loop fusion* transformation.

```
L1   for I1 = -3,-1
L2    for I2 = 1,3
S1     A[-I1,I2] = ...
S2     ... = A[-I1 + 1,I2 - 1]
```

Figure 5.5(b). Sample code: *loop fusion* and *loop reversal.*

```
L2   for I2 = 1,3
L1    for I1 = 1,3
S1     A[I1,I2] = ...
S2     ... = A[I1 + 1,I2 - 1]
```

Figure 5.5(c). Sample code: *loop fusion* and *loop interchange.*

upon completion of the transformation sequence. In summary, the legality criteria for loop fusion is:

Preconditions

Invariant

$\boxed{\text{TRUE}}$

Explicit

$\boxed{\texttt{can\_adjust}(B_{\Theta_1}, B_{\Theta_2})}$

Nesting

$\boxed{\text{TRUE}}$

5.2.2 Effect on the Dependence Vector Set D

Modifications to the distance vector set $D^{(x)}$ were discussed above in the context of loop fusion legality. There are four partitions to the distance vector set after loop fusion: $D^{(x-1)}_{\Theta_1,\Theta_1}$, $D^{(x-1)}_{\Theta_2,\Theta_2}$, $D^{(x)}_{\Theta_1,\Theta_2}$, and $D^{(x)}_{\Theta_2,\Theta_1}$. The distance vector set after loop fusion is the union of these four partitions with $D^{(x)}_{\Theta_2,\Theta_1}$ negated. That is,

$$\boxed{D^{(x)} = D^{(x-1)}_{\Theta_1,\Theta_1} \cup D^{(x-1)}_{\Theta_2,\Theta_2} \cup D^{(x)}_{\Theta_1,\Theta_2} \cup -(D^{(x)}_{\Theta_2,\Theta_1})}$$

5.2.3 Effect on Source Code Mapping

The composability property states that any unimodular transformation can be applied at any time. Therefore, the loop nest Θ_1 may have a sequence of transformations applied to it prior to the loop fusion transformation (T_{Θ_1}). The loop nest Θ_2 may also have a sequence of

transformations applied to it prior to loop fusion (T_{Θ_2}). Also, the loop bound sets B_{Θ_1} and B_{Θ_2} may need "adjustment". In any case, assume that after loop fusion, the statements originally from Θ_1 will have a different transformation matrix associated with them (T_{Θ_1}) than the statements originally from Θ_2 (T_{Θ_2}). The implementation of loop fusion must account for both T_{Θ_1} and T_{Θ_2} in the transformed source code. In other words, we have two transformation matrices for one loop nest after the loop fusion transformation.

The final source code mapping proceeds in two steps. The assignment statements that were initially in Θ_1 are mapped using T_{Θ_1}. Similarly, the assignment statements that were initially in Θ_2 are mapped using T_{Θ_2}.

As an example, the source code in Figure 5.6(a) cannot be fused in its current form because the loop bounds for the inner loops do not exactly match (loop L_2' is the reversal of loop L_2). However, `can_adjust`$(B_{\Theta_1}, B_{\Theta_2})$ is *true*, because the outer loop and inner loop of Θ_1 have the same number of iterations as the outer loop and inner loop of Θ_2, respectively. If reversal is applied to the inner loop of Θ_2, then the source code in Figure 5.6(b) is obtained. Then the loops can be fused as shown in Figure 5.6(c), resulting in semantically equivalent source code.

The transformation matrix for Θ_1 is:

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

where Θ_1 =(S1), and the transformation for Θ_2 is:

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

where Θ_2 =(S2) and T_{Θ_2} reflects the reversal of the inner loop. After loop fusion, T_{Θ_1} remains associated with Θ_1 and is used to determine the form of the assignment statements in Θ_1. Similarly, T_{Θ_2} remains associated with Θ_2 and is used to determine the form of the assignment statements in Θ_2. Finally, *either* T_{Θ_1} or T_{Θ_2} is used to map the loop bounds. That is, mapping the loop bounds for Θ_1 using T_{Θ_1} results in the same loop bounds as mapping the loop bounds for Θ_2 using T_{Θ_2}.

```
L1   for I1 = 1, m
L2    for I2 = 1, n
S1      A[I1] = ...
L1'   for I3 = 1, m
L2'    for I4 = -n, -1
S2      ... = A[-I4 - 1]
```

Figure 5.6(a). Sample code: before *adjustment*

```
L1   for I1 = 1, m
L2    for I2 = 1, n
S1      A[I1] = ...
L1'   for I3 = 1, m
L2'    for I4 = 1, n
S2      ... = A[I4 - 1]
```

Figure 5.6(b). Sample code: after *adjustment*

```
L1   for I1 = 1, m
L2    for I2 = 1, n
S1      A[I1] = ...
S2      ... = A[I2 - 1]
```

Figure 5.6(c). Sample code: after *adjustment* and *loop fusion*

In summary, the transformation matrix for the assignment statements originally in Θ_1 is $T_{\Theta_1}^{(x)}$ and the transformation matrix for the assignment statements originally in Θ_2 is $T_{\Theta_2}^{(x)}$.

Θ_1 statements:	$T_{\Theta_1}^{(x)} = T_{\Theta_1}^{(x-1)}$
Θ_2 statements:	$T_{\Theta_2}^{(x)} = T_{\Theta_2}^{(x-1)}$

CHAPTER 6

Integration of the Extended Loop Transformations

Algorithms that use the kernel and extended set of transformations are broadly classified as either machine-independent or machine-dependent. Machine-independent algorithms determine a transformation matrix T that meets some general goal, such as parallelism or data locality. Machine-dependent algorithms determine a transformation matrix T that optimizes source code for a particular architecture, such as a vector or shared-memory machine.

The kernel unimodular transformations are currently used to obtain parallelism, and in some cases data locality, on source code in perfect loop nests. In order to provide context and motivation for developing algorithms to integrate the extended loop transformations, this chapter begins by reviewing the existing algorithms that use the kernel unimodular transformations to obtain coarse- and fine-grain parallelism.

Next, new algorithms that we developed are described that use the kernel and extended sets of transformations to obtain new goals or that enable the application of transformations to a broader range of source code structures. First, the wavefront algorithm that achieves fine-grain parallelism is modified so that it is applicable to imperfect-only-child loop nests, in addition to perfect loop nests. Second, the extended transformations are used to expand the machine-independent goals: first, by finding parallelism where the kernel unimodu-

lar transformations fail; and second, by obtaining data locality within the matrix-based framework. Finally, the extended transformations are used to obtain machine-dependent optimization. Specifically, two algorithms are developed for source code executing on vector and shared-memory machines. All of the algorithms that use the extended set of loop transformations can be applied to source code with both perfect and imperfect-only-child loop nests.

Both Banerjee [3], and Wolf and Lam [4] have developed algorithms that yield coarse-grain parallelism in some cases, and fine-grain parallelism in all cases of perfect loop nests. Using their algorithms, coarse-grain parallelism may not be obtainable with the kernel unimodular transformations. However, we have developed an algorithm that uses the extended set of transformations and may allow parallelization of the outermost loops in cases where the kernel set of transformations fail.

Similarly, fine-grain parallelism refers to the parallelism of the innermost loops of a perfect, n-deep loop nest. Wolf and Lam show that $n-1$ degrees of parallelism can always be obtained from a n-deep perfect loop nest. A new algorithm is discussed that is a modification of the original algorithm for fine-grain parallelism and works on imperfect loop nests. Specifically, if assignment statements are nested at q unique depths in a loop nest, then the algorithm finds a minimum of $n-q$ degrees of parallelism. As examples, q is one for a perfect loop nest because all statements are nested at one level, and q is greater than one for any imperfect loop nest.

Data *reuse* is defined as using the same datum in different iterations of a loop nest. Data *locality* occurs if reused data remains in the same memory hierarchy level between uses. An algorithm is presented that improves data locality using the transformations from the extended set, namely loop fission and fusion.

Next, two architecture-specific algorithms are introduced that optimize source code using the extended set of loop transformations. Specifically, an algorithm is given that optimizes source code for a vector machine using loop normalization, fission, permutation, and blocking. Finally, an algorithm for coarse-grain parallelism on a shared-memory machine is discussed, and uses loop normalization, fusion, permutation, and collapsing. The objective

of the vector machine algorithm is to minimize work performed in each loop and move the parallel loops innermost. In contrast, the objective of the shared memory machine algorithm is to maximize the work performed in each loop and move the parallel loops outermost. The development of these algorithms shows that the extended set of loop transformations can be composed to achieve both machine-independent and machine-dependent compiler goals and can be applied to a wide variety of source code structures.

6.1 Previously Developed Algorithms

This section discusses two existing algorithms that are applied to perfect loop nests. The first algorithm obtains coarse-grain parallelism on a doubly-nested loop, but can be generalized to a n-nested loop. The second algorithm obtains fine-grain parallelism on a n-nested loop.

Loop L_i can be parallelized if all of the distance vectors $\vec{d}$ in the distance vector set D have specific properties. If each distance vector $\vec{d}$ is in the form $(d_1, d_2, \ldots, d_n)$, where n is the loop nest depth and $\vec{d} \succ \vec{0}$ means that distance vector $\vec{d}$ is lexicographically positive, then loop L_i can be parallelized if:

$$(\forall \vec{d} : \vec{d} \in D : ((d_1, \ldots, d_{i-1}) \succ \vec{0}) \vee (d_i = 0)). \tag{6.1}$$

That is, loop L_i can be parallelized if each partial distance vector preceding location i is lexicographically positive or each distance at location i is 0. In the following discussion, *disj1* refers to the first disjunct of Expression (6.1), $((d_1, \ldots, d_{i-1}) \succ \vec{0})$ and *disj2* refers to the second disjunct of Expression (6.1) $(d_i = 0)$.

As an example, suppose that D contains only one distance vector: $D = \{(0, 2, -2)\}$. The following table shows the parallelism in the source code based upon *disj1* and *disj2*:

	$d_i = 0$	$(d_1, \ldots, d_{i-1}) \succ \vec{0}$	Parallel
L_1	yes	not applicable	yes
L_2	no	no	no
L_3	no	yes	yes

6.1.1 Coarse-grain parallelism for perfect loop nests using the kernel transformation set

A distance vector $\vec{d}$ for a doubly-nested loop is in the form (d_1, d_2), where d_1 corresponds to the outer loop (L_1) and d_2 corresponds to the inner loop (L_2). Parallelization of the outer loop is equivalent to finding a transformation matrix T, such that $d_1 = 0$ for all of the distance vectors in the transformed distance vector set $T \times \vec{d} \in D$. Satisfying this requirement is equivalent to satisfying *disj2* from Expression (6.1). Note that *disj1* is irrelevant in this case since $i - 1 = 0$.

For a doubly-nested loop, the transformation matrix T is in the form:

$$\begin{bmatrix} T[1,1] & T[1,2] \\ T[2,1] & T[2,2] \end{bmatrix}$$

Therefore, finding T that parallelizes the *outermost* loop is equivalent to finding $T[1,1]$ and $T[1,2]$, such that $T[1,1] \times d_1 + T[1,2] \times d_2 = 0$ for all $\vec{d} \in D$. The values of $T[2,1]$ and $T[2,2]$ are later calculated to make T unimodular. That is, both $T[2,1]$ and $T[2,2]$ are integers and are calculated such that $|det(T)| = 1$.

Wolf and Lam [4] describe coarse-grain parallelism as choosing the transformation matrix T with first row $\vec{t_1}$ such that $\vec{t_1} \times \vec{d} = 0$ for all dependence vectors $\vec{d}$. Banerjee [3] describes an algorithm for parallelizing the outermost loop of a doubly-nested loop such that the transformed source code is semantically equivalent to the original source code; the outer loop can be executed in parallel; and the iteration count of the outermost loop is maximized. Transformation matrix T results in $d_1 = 0$ for all transformed distance vectors

in D. Algorithm **kernel_coarse** obtains coarse-grain parallelism on a perfect, 2-deep loop nest using the kernel transformations and is given in Figure 6.1.

Algorithm kernel_coarse
Finds a 2×2 transformation matrix T that parallelizes the outermost loop of a doubly-nested loop, if one exists.

Input:

D /**the distance vector set**/

Output:

T /**the 2×2 transformation matrix**/

Procedure:

1. Move the loop with the highest iteration count to the outermost position using permutation.
2. Determine if all $\vec{d} \in D$ are in *basis* form $\alpha(a_1, a_2)$, where α is a positive integer and (a_1, a_2) is lexicographically positive. If all of $\vec{d} \in D$ cannot be placed in this form, then terminate. Else, continue.
3. Find $T[2,1], T[2,2]$ and $g = \text{GCD}(a_1, a_2)$ such that $g = a_1 \times T[2,1] + a_2 \times T[2,2]$, where $T[1,1] = a_2/g$ and $T[1,2] = -a_1/g$.

Figure 6.1. Algorithm to obtain coarse-grain parallelism using the kernel transformations of perfect loop nests.

Note that a transformation matrix T *may not* exist that results in coarse-grain parallelism. That is, Banerjee's algorithm terminates (Step 2) if the distance vectors cannot be placed in *basis* form. The extended algorithm discussed in Section 6.4 shows how to increase the chances of finding a basis form for all distance vectors in the distance vector set.

6.1.2 Fine-grain parallelism for perfect loop nests using the kernel transformation set

This section describes an algorithm that parallelizes the inner $n-1$ loops of a n-deep, perfect loop nest using the kernel set of loop transformations. The approach is known as the *wavefront* transformation [4, 24], and is named **kernel_fine** in this dissertation.

Two important proofs developed by Wolf and Lam [4] are critical to the applicability of the wavefront transformation. The first shows that any lexicographically positive distance vector can be made fully permutable. The second shows that any perfect, n-deep loop nest with a fully permutable distance vector set D can be transformed to have $n-1$ degrees of parallelism. The proofs are paraphrased below, and the reader is referred to Wolf and Lam [4] for more details.

All distance vectors in the initial distance vector set are lexicographically positive, where a lexicographically positive $\vec{d}$ has at least one positive distance d_k. The second disjunct of Expression (2.1) (definition of fully permutable) requires that distances d_i through d_j must be non-negative, if loops L_i through L_j are fully permutable. Any negative distance in the range d_i through d_j can be made non-negative via skewing by the positive distance d_k. That is, any negative distances between loops L_i and L_j can be made non-negative satisfying the second disjunct of Expression (2.1).

Step 1 of Algorithm **kernel_fine** transforms a lexicographically positive distance vector set D into a fully permutable distance vector set $D^{(1)}$. Transforming a lexicographically positive distance vector set into a fully permutable distance vector set involves skewing the inner loops by the positive distance d_k in each distance vector $\vec{d}$. As an example, the transformation matrix $T^{(1)}$ for a 4-deep loop nest is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \mu_{2,1} & 1 & 0 & 0 \\ \mu_{3,1} & \mu_{3,2} & 1 & 0 \\ \mu_{4,1} & \mu_{4,2} & \mu_{4,3} & 1 \end{bmatrix}$$

where $\mu_{i,j} = \lceil \max(-d_j/d_i) \rceil$, such that $d_i > 0$ and $\vec{d} \in D$.

After transforming the distance vectors to fully permutable form, loops L_2 through L_n are parallelized by satisfying *disj1* from Expression (6.1) (definition of parallelism). That is, loops L_2 through L_n are parallelized by ensuring that the distances associated with loop L_1 are strictly positive ($\forall \vec{d} : \vec{d} \in D : d_1 > 0$). In Expression (6.1), *disj1* states that loop L_i can be parallelized if ($\forall \vec{d} : \vec{d} \in D : (d_1, \ldots, d_{i-1}) \succ \vec{0}$). In an n-nested loop, if we show that the first distance is greater than zero ($d_1 > 0$), then *disj1* is satisfied.

In fully permutable form, all distances in a distance vector set are non-negative. The wavefront transformation skews the innermost loop L_n of a n-nested loop by each of the outer loops L_1 through L_{n-1}. Therefore, all distances associated with the innermost loop d_n are strictly positive. As a final step in the wavefront transformation, the innermost loop is moved outermost. Thus, rather than d_n being positive for all $\vec{d} \in D$, d_1 is positive for all $\vec{d} \in D$, which satisfies *disj1* from Expression (6.1). The wavefront transformation matrix T skews the innermost loop by each of the outer loops and then moves the innermost loop to the outermost position. The wavefront matrix for a 4-deep loop is in the form:

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

Algorithm `kernel_fine` obtains $n-1$ degrees of parallelism from a n-deep, perfect loop nest. Step 1 transforms all lexicographically positive distance vectors in the distance vector set so that they are fully permutable. Step 2 skews the innermost loop by each of the outer loops. Finally, Step 3 moves the innermost loop to the outermost position using loop permutation.

6.2 New Algorithms

This section introduces several new algorithms that make use of the extended and kernel set of matrix-based loop transformations to address different objectives of a parallel compiler. The first algorithm is a modified version of the wavefront transformation and can be applied to imperfect loop nests. The second algorithm achieves coarse-grain parallelism in

Algorithm kernel_fine
Finds a $n \times n$ transformation matrix T that parallelizes the innermost $n-1$ loops of a n-deep perfect loop nest.

Input:

D /**the initial distance vector set**/

Output:

T /**the $n \times n$ transformation matrix**/
D' /**the transformed distance vector set**/

Procedure:

1. /**Make all $\vec{d} \in D$ fully permutable by skewing**/
   For all i, $2 \leq i \leq n$
   For all j, $1 \leq j < i$
   skew(L_i, L_j,**max**$\lceil -d_j/d_i \rceil$), such that $d_i > 0$
   Endfor
   Endfor

2. /**Skew the innermost loop L_n by each of the outer loops**/
   For all k, $1 \leq k < n$
   skew(L_n, L_k, 1)
   Endfor

3. /**Permute the innermost loop L_n to the outermost position**/
   permute(L_1, L_n)

Figure 6.2. Algorithm to obtain fine-grain parallelism using the kernel transformations on perfect loop nests.

cases where the kernel unimodular transformations cannot and uses loop fission to reduce the cardinality of the dependence vector set D. The dependence vectors in D form the constraints on possible coarse-grain parallelism. Thus, reducing the number of constraints (i.e., reducing the cardinality of D) results in more potential parallelism.

Next, an algorithm is presented that obtains data locality. Data locality has not previously been addressed with respect to matrix transformations. However, loop blocking was used by Wolf [24] to obtain data locality external to the matrix-based framework. We use loop fission and fusion to obtain locality within the matrix-based model.

Finally, two architecture-specific algorithms are given that transform source code for a given machine. Specifically, an algorithm for vector machines is discussed, as is an algorithm for a shared-memory system.

6.3 Fine-grain parallelism for imperfect-only-child loop nests using the kernel transformation set

This section describes a modification to Algorithm **kernel_fine** that allows it to be applied to imperfect loop nests. The algorithm, named **mod_kernel_fine**, applies the wavefront transformation to *partitions* or *regions* of the nested loop. We show that if m statements $(S_1, \ldots, S_m)$ are nested at q unique levels in a n-deep loop, then the source code can be transformed such that it has a minimum of $n - q$ degrees of parallelism.

As an example, the source code shown in Figure 6.3 has an assignment statement S_1 nested i-deep, and several assignment statements $(S_2, \ldots, S_m)$, nested n-deep. Therefore, there are two unique levels at which assignment statements are nested ($q = 2$), and the source code can be transformed such that $n - q = n - 2$ degrees of parallelism result.

```
L1      for I1 = ...
L2        for I2 = ...
 :          :
Li          for Ii = ...
S1            A[I1, ..., Ii] = ...
Lj            for Ij = ...
Lj+1            for Ij+1 = ...
 :                :
Ln                for In = ...
S2                  B[I1, ..., In] = ...
 :                  :
Sm                  C[I1, ..., In] = ...
```

Figure 6.3. Sample code: n-deep imperfect-only-child loop nest.

Algorithm **mod_kernel_fine**, given in Figure 6.4, is similar to Algorithm **kernel_fine** except that it parallelizes within regions that are defined by imperfectly nested assignment statements. Using the source code shown in Figure 6.3 as an example, the Algorithm **mod_kernel_fine** first parallelizes loops L_1 through L_i using the wavefront transformation. Next, loops L_j through L_n are parallelized using the wavefront transformation. The source code is parallelized in regions because the legality condition of loop permutation does not allow interchange of loops that are not perfectly nested within each other. Any two loops in the range L_1 through L_i can be interchanged as needed, as can any two loops in the range L_j through L_n. However, loops cannot be permuted *between* the two ranges.

The first step of Algorithm **mod_kernel_fine** is to transform all distance vectors in the dependence vector set into fully permutable form using loop skewing. Loop skewing is legal regardless of the nesting structure of the source code. That is, loop skewing can be legally applied across imperfectly nested loops.

A significant result of the skewing transformation is that any partial distance vector is also fully permutable. As an example, if the distance vector $\vec{d} = (d_1, \ldots, d_n)$ is fully permutable, then so is any $(d_i, \ldots, d_j)$, where $1 \leq i \leq j \leq n$. All of the distances are non-negative in a fully permutable distance vector, and therefore, any partial distance vectors $(d_i, \ldots, d_j)$ are either the zero vector or are strictly positive.

If the distances in a region are all zero ($(d_i, \ldots, d_j) > \vec{0}$) then all loops L_i through L_j can be parallelized by *disj2* of Expression (6.1). If the distance vectors are strictly positive, then the inner loop in the region can be skewed by each of the outer loops in the region and moved outermost. This strategy is similar to the wavefront transformation, however, it is applied across a region of a n-deep loop nest bounded by nested assignment statements, rather than all n loops.

The motive for the application of region wavefronting is to maximize parallelism despite the constraints imposed by the legality of loop permutation. That is, loops are permuted only within a region. Maximal parallelism within a n-deep imperfect-only-child loop nest is guaranteed because maximum parallelism is obtained within each region bound by an imperfectly nested assignment statement.

Algorithm mod_kernel_fine
Finds a $n \times n$ transformation matrix T that parallelizes a minimum of $n - q$ loops of a n-deep imperfect-only-child loop nest, where q is the number of unique depths at which assignment statements are nested.

Input:

D /***the initial distance vector set***/
$depth[1 \ldots q]$ /***ordered set of unique depths at which assignment statements are nested***/

Output:

T /***the $n \times n$ transformation matrix***/
D' /***the transformed distance vector set***/

Procedure:

1. /***Make all $\vec{d} \in D$ fully permutable by skewing***/
   For all i, $2 \leq i \leq n$
   For all j, $1 \leq j < i$
   skew(L_i, L_j,**max**$\lceil -d_j/d_i \rceil$), such that $d_i > 0$
   Endfor
   Endfor

2. (A) /***Initialize the counters***/
   $depth\_index = 1$
   $start\_depth = 1$

   (B) /***For each partition of the loops***/
   Repeat
   $end\_depth = depth[depth\_index]$

   (C) /***Skew within regions***/
   For all k, $start\_depth \leq k < end\_depth$
   skew(L_{end_depth}, L_k, 1)
   Endfor

   (D) /***Permute within regions***/
   permute(L_{start_depth}, L_{end_depth})

   (E) /***Increment the counters***/
   $depth\_index$ ++
   $start\_depth = end\_depth + 1$
   Until $depth\_index > q$

Figure 6.4. Algorithm to obtain fine-grain parallelism using the kernel transformations on imperfect-only-child loop nests.

As an example, Algorithm **mod_kernel_fine** is applied to the source code given in Figure 6.5. The initial distance vector set for the source code is $D = \{(0,0,1,-1),(0,0,0,1)\}$. Assignment statement S_1 is nested 2-deep and assignment statements S_2 through S_5 are nested 4-deep. An array, named *depth* stores the ordered set of unique depths at which assignment statements are nested. Thus, $q = 2$ and $depth[1 \ldots q] = [2,4]$.

L_1 **for** $I_1 = 1, ub_1$
L_2 **for** $I_2 = 1, ub_2$
S_1 $A[I_1, I_2] = \ldots$
L_3 **for** $I_3 = 1, ub_3$
L_4 **for** $I_4 = 1, ub_4$
S_2 $B[I_1, I_2, I_3, I_4] = \ldots$
S_3 $B[I_1, I_2, I_3 - 1, I_4 + 1] = \ldots$
S_4 $C[I_1, I_2, I_3, I_4] = \ldots$
S_5 $C[I_1, I_2, I_3, I_4 - 1] = \ldots$

$/ * * D = \{(0,0,1,-1),(0,0,0,1)\} * * /$

Figure 6.5. Sample code: initial code before application of **mod_kernel_fine**.

Step 1 transforms the distance vector set such that all $\vec{d} \in D$ are fully permutable. Specifically, loop L_4 is skewed by loop L_3 and the resulting distance vector set is $D^{(1)} = \{(0,0,1,0),(0,0,0,1)\}$. Note that all of the distances in the transformed distance vector set are non-negative.

Step 2(A) initializes the value for indexing the array *depth*, and Step 2(B) iterates through *depth*. Each region of the loop nest, defined by the depths of the assignment statements is traversed. The first region to which the transformation is applied is loops L_1 through L_2. That is, $end\_depth = depth[1] = 2$. The next iteration of Step 2(B) processes loops L_3 through L_4. That is, $start\_depth = end\_depth + 1 = 3$ and $end\_depth = depth[2] = 4$.

Steps 2(C) and 2(D) are equivalent to Steps 2 and 3 from Algorithm **kernel_fine**, respectively, except that the wavefront transformation is applied to regions of the n-deep loop nest, rather than all n loops. The skewing transformation (Step 2(C)) skews loop

L_2 by loop L_1. However, since d_1 and d_2 are zero for both distance vectors, the resulting distance vector set is $D^{(2(C))} = \{(0,0,1,0),(0,0,0,1)\}$. Next, loop L_4 is skewed by loop L_3 resulting in $D^{(2(C))} = \{(0,0,1,1),(0,0,0,1)\}$. Finally, loops L_3 and L_4 are permuted resulting in $D^{(2(D))} = \{(0,0,1,1),(0,0,1,0)\}$. Loops L_1, L_2, and L_4 are parallelized using the definition for parallelism given in Expression (6.1) as shown in Figure 6.6.

L_1 **forall** $J_1 = 2, ub_2 + ub_1$
L_2 **forall** $J_2 = \mathbf{max}(1, J_1 - ub_2), \mathbf{min}(ub_1, J_1 - 1)$
S_1 $A[J_2, J_1 - J_2] = \ldots$
L_3 **for** $J_3 = 3, ub_4 + 2ub_3$
L_4 **forall** $J_4 = \mathbf{max}(1, \lfloor \frac{J_3 - ub_4}{2} \rfloor), \mathbf{min}(ub_3, \lfloor \frac{J_3 - 1}{2} \rfloor)$
S_2 $B[J_2, J_1 - J_2, J_4, J_3 - 2J_4] = \ldots$
S_3 $B[J_2, J_1 - J_2, J_4 - 1, J_3 - 2J_4 + 1] = \ldots$
S_4 $C[J_2, J_1 - J_2, J_4, J_3 - 2J_4] = \ldots$
S_5 $C[J_2, J_1 - J_2, J_4, J_3 - 2J_4 - 1] = \ldots$

$/ * * D' = \{(0,0,1,1),(0,0,1,0)\} * * /$

Figure 6.6. Sample code: transformed code after application of **mod_kernel_fine**.

6.4 Coarse-grain parallelism for imperfect-only-child loop nests using the extended transformation set

Figure 6.7 gives an algorithm, named **ext_coarse**, that uses loop fission within a matrix-based framework to obtain parallelism where kernel unimodular transformations alone would not. In addition to using loop fission, the permutation step of Algorithm **kernel_coarse** is modified in Algorithm **ext_coarse**. The modification is due to the illegality of loop permutation for imperfectly nested loops.

Specifically, if there are *any* statements between loops L_1 and L_2, then loops L_1 and L_2 cannot be permuted. However, if there are *no* statements between loops L_1 and L_2, then the loop with the highest iteration count is moved outermost.

Algorithm **kernel_coarse** (Figure 6.1) terminated if any $\vec{d} \in D$ did not have the same *basis* $\alpha(a_1, a_2)$. The distance vectors that cannot be placed in *basis* form are further examined in Algorithm **ext_coarse**. If possible, the statements causing the *non-basis* distances are separated into adjacent Ψ-blocks by loop fission. Chapter 5 showed that strongly connected components (SCC) of the dependence graph cannot be divided by loop fission. Consistent with that constraint, Algorithm **ext_coarse** does not divide SCC's of the dependence graph. However, if the *non-basis* distances are not in a strongly connected component, then the statements causing the distance are placed in separate Ψ-blocks. Thus, the distance vectors preventing parallelism that were originally in the distance vector set D become loop-independent and are consequently ignored.

Loop fission can be applied to both perfect and imperfect loop nests, and thus, an algorithm that uses loop fission can be applied to both perfect and imperfect loop nests. The source code given in Figure 6.8 cannot be parallelized using Algorithm **kernel_coarse**, but can be parallelized using Algorithm **ext_coarse**. We want to parallelize L_1 because the iteration count is significantly greater than the iteration count of L_2. The strongly connected components of the dependency graph are $\{(S_1, S_2), (S_3), (S_4)\}$.

The distance vector (0,3) can be placed in *basis form* 3(0,1), where $\alpha = 3$ and the *basis* is (0,1). The distance vector (0,4) can be also be placed in basis form 4(0,1), where $\alpha = 4$. However, the distance vector (1,2) cannot be placed in basis form. That is, there is no positive integer α that, when multiplied by (0,1), equals (1,2). Therefore, the distance vector (1,2) causes termination of Algorithm **kernel_coarse**. Algorithm **ext_coarse** determines that S_3 and S_4 are not in the same strongly connected component and the statements are placed in adjacent loop nests by loop fission, as shown in Figure 6.9. The outer loops of both Ψ-blocks are parallelized in the source code in Figure 6.9 because no dependency prevents it.

Algorithm ext_coarse
Finds a 2×2 transformation matrix T that parallelizes the outermost loop of a doubly-nested loop, if one exists.

Input:

D /**the distance vector set**/
$SCC(D)$ /**strongly connected components of D**/

Output:

$T(\Psi_i)$ /**the 2×2 transformation matrix**/

Procedure:

1. If there are no statements between loops L_1 and L_2, then move the loop with the highest iteration count to the outermost position using permutation.

2. Determine if all $\vec{d} \in D$ can be put in *basis* form $\alpha(a_1, a_2)$, where α is a positive integer and (a_1, a_2) is lexicographically positive.

   (A) If any $\vec{d} \in D$ in *non-basis* form is caused by statements in the same strongly connected component, then terminate;

   (B) If any $\vec{d} \in D$ in *non-basis* form is caused by statements in different strongly connected components, then place the statements in separate Ψ-blocks;

   (C) If all $\vec{d} \in D$ are in *basis* form, then continue.

3. For each Ψ-block, find $T[2,1], T[2,2]$ and $g = \text{GCD}(a_1, a_2)$ such that $g = a_1 \times T[2,1] + a_2 \times T[2,2]$, where $T[1,1] = a_2/g$ and $T[1,2] = -a_1/g$.

Figure 6.7. Algorithm to obtain coarse-grain parallelism using the extended transformations on imperfect-only-child loop nests.

```
L1   for I1 = 1,1000
L2    for I2 = 1,6
S1     A[I1,I2] = B[I1,I2 - 3]
S2     B[I1,I2] = A[I1,I2 - 4]
S3     C[I1,I2] = ...
S4     C[I1 - 1,I2 - 2] = ...

/**D = {(0,3),(0,4),(1,2)}**/
```

Figure 6.8. Sample code: parallelization not allowed by kernel unimodular transformations.

```
L1   forall I1 = 1,1000
L2    for I2 = 1,6
S1     A[I1,I2] = B[I1,I2 - 3]
S2     B[I1,I2] = B[I1,I2 - 4]
S3     C[I1,I2] = ...
L3   forall I1 = 1,1000
L4    for I2 = 1,6
S4     C[I1 - 1,I2 - 2] = ...
```

Figure 6.9. Sample code: parallelization after loop fission.

6.5 Data locality for imperfect-only-child loop nests using the extended transformation set

This section discusses an algorithm that uses transformations from the extended transformation set to obtain data locality. Wolf [24] discusses the use of loop blocking (tiling) to obtain data locality, where loop blocking is considered as a separate transformation outside of the unimodular model.

We present an algorithm that uses loop fission and fusion to obtain data locality. The legality conditions for loop fission and fusion are not dependent upon source code structure. Correspondingly, this algorithm is applicable to both perfect and imperfect-only-child loop nests. Chapter 5 discussed the legality criteria for loop fission and fusion, but did not discuss

the context in which the two transformations should be applied. Algorithm **ext_locality** shows when to apply loop fission and fusion in order to obtain data locality.

Reuse was earlier defined as the use of the same datum on different iterations of a loop nest, and *locality* refers to the reused data remaining in the same memory hierarchy level (e.g. cache) between uses. The source code in Figure 6.10 shows that reuse does not guarantee locality, especially if the cache is small. That is, A[1] is *reused* in each iteration of L_1, however, A[1] is *local* only if ub_2 is smaller than the cache size. Data reuse is a property of the source code and is not dependent upon the loop structure or machine. In contrast, locality is dependent upon machine characteristics, such as cache size. Thus, increasing data locality results in a decreased number of memory accesses and improved use of the cache.

L_1 **for** $I_1 = 1, ub_1$
L_2 **for** $I_2 = 1, ub_2$
S_1 A$[I_2] = \ldots$

Figure 6.10. Sample code: showing reuse and locality.

Section 2.1 discussed the representation of array subscript expressions using *array index functions* and *uniformly generated sets*. A uniformly generated set is an equivalence class, such that any two members of the set have the same array index function H and refer to the same array. Therefore, our objective is to determine the uniformly generated sets with maximum cardinality or simply to form the largest possible groupings of similar references to the same array. Loop fission and fusion are used to separate and/or combine groupings of statements and provide the basis for the algorithm that improves locality.

As an example, all assignment statements shown in Figure 6.11 reference the same array A. However, the array index function of S_1 is the same as S_4. Similarly, the array index function of S_2 is the same as S_3. Thus, the greatest locality results from grouping S_2 and S_3 into one equivalence class, and S_1 and S_4 into another.

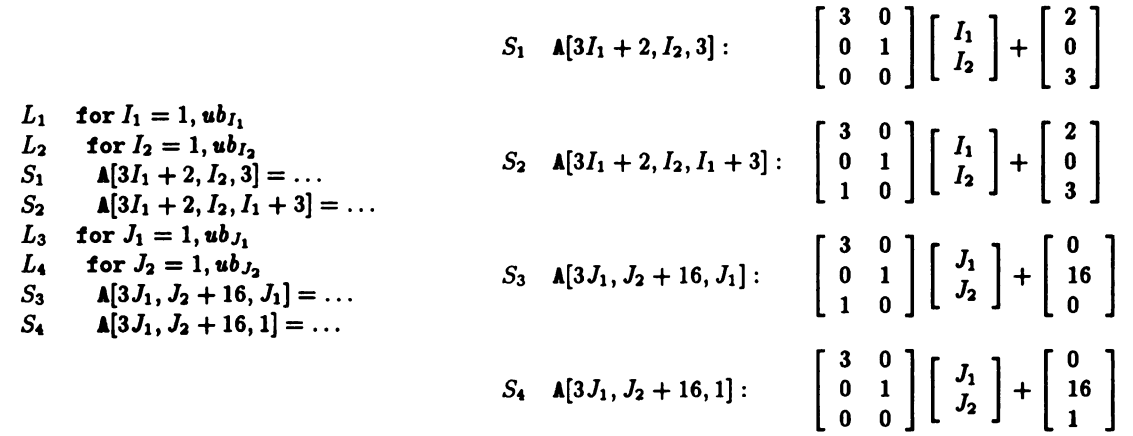

Figure 6.11. Sample code: statement groupings for data locality $\{(S_1, S_4), (S_2, S_3)\}$.

Algorithm **ext_locality**, shown in Figure 6.12, obtains data locality using loop fission and fusion. The objective of the algorithm is to partition a set of statements $(S_1, \ldots, S_n)$ into the largest possible sets, such that the statements in the partitions have similar accesses to the same variable. The input to the algorithm is two adjacent Θ-blocks (loop nests). The algorithm uses loop fusion to combine similar accesses to the same variable and loop fission to separate accesses to different variables. Note that the statements in different Θ-blocks are only combined if the loop bound sets can be adjusted and the statements are separated only if they are not in the same Π-block, regardless of the perfect or imperfect nesting structure of the source code.

The source code shown in Figure 6.13 becomes the source code shown in Figure 6.14 after application of Algorithm **ext_locality**. The partitions are initialized to the strongly connected components of the source code $\{(S_1), (S_2, S_6), (S_3), (S_4, S_7), (S_5)\}$, then through pairwise comparison the assignment statements are combined into larger sets. The transformation results in the following partition of the assignment statements: $\{(S_1, S_5), (S_2, S_3, S_6), (S_4, S_7)\}$ because each set has similar references to the same variable as indicated by the array index functions.

Algorithm ext_locality
Partitions the assignment statements in two adjacent Θ-blocks into groups with increased data locality.

Input:

B_{Θ_1}	/***loop bound set for the first loop nest***/
B_{Θ_2}	/***loop bound set for the second loop nest***/
D_{Θ_1,Θ_1}	/***the distance vector set for the first loop nest***/
D_{Θ_2,Θ_2}	/***the distance vector set for the second loop nest***/
S_{Θ_1}	/***set of assignment statements in the first loop nest***/
S_{Θ_2}	/***set of assignment statements in the second loop nest***/

Output:

$\Psi_1, \ldots, \Psi_m$ /***partitions of* S_{Θ_1} *and* S_{Θ_2}**/

Procedure:

1. If **can_adjust**(B_{Θ_1},B_{Θ_2})

   (A) Initialize the partitions $\Psi_1, \ldots, \Psi_m$ to the strongly connected components of the assignment statements.

   (B) For each pair of assignment statements S_i and S_j:

   (i) If S_i and S_j are in separate SCC's

   (a) If S_i and S_j have the same array index function for the same variable then combine the partitions containing S_i and S_j.

Figure 6.12. Algorithm to improve data locality using the extended transformations on imperfect-only-child loop nests.

L_1 **for** $I_1 = 1, ub_1$
S_1 $G[I_1] = \ldots$
L_2 **for** $I_2 = 1, ub_2$
S_2 $A[I_1, I_2] = B[I_1 - 1, I_2]$
S_3 $var_1 = B[I_1 - 1, I_2 - 1]$
S_4 $C[I_1, 2I_2] = D[I_1, 2I_2]$

L_3 **for** $J_1 = 1, ub_1$
S_5 $\ldots = G[J_1 - 2]$
L_4 **for** $J_2 = 1, ub_2$
S_6 $B[J_1, J_2] = A[J_1 - 1, J_2]$
S_7 $D[J_1 - 1, 2J_2] = C[J_1 - 1, 2J_2]$

/ * **SCC** = $\{(S_1), (S_2, S_6), (S_3), (S_4, S_7), (S_5)\}$ * /

Figure 6.13. Sample code: before application of data locality algorithm.

Comments on coarse-grain parallelism and data locality algorithms. Intuitively, Algorithm **ext_coarse** and Algorithm **ext_locality** could be merged into one algorithm that combines both parallelism and data locality goals. However, these two goals often conflict. That is, source code that is transformed to obtain maximum parallelism may have poor data locality, and vice-versa.

As an example, the source code shown in Figure 6.15(a) has two singly-nested loops. In the form shown in Figure 6.15(a), both L_1 and L_2 can be parallelized since there is no data dependence that prevents parallelization. However, statements S_1 and S_2 have the same array index functions and could be combined if data locality were a priority. After the loops are combined, as shown in Figure 6.15(b), the data locality is improved but the enclosing loop can no longer be parallelized due to the data dependence between statements S_1 and S_2.

This simple example shows that parallelism goals and data locality goals may conflict, and blindly combining Algorithms **ext_coarse** and **ext_locality** may result in an algorithm that obtains *neither* parallelism nor data locality. A possible approach for a combined algorithm is to assign a weight to both parallelism and data locality goals. When a potential conflict arises, then the goal with the greater weight is met at the expense of the goal with

```
L1   for K1 = 1, ub1
S1    G[K1] = ...
S5    ... = G[K1 - 2]

L1   for K1 = 1, ub1
L2    for K2 = 1, ub2
S2     A[K1, K2] = B[K1 - 1, K2]
S3     var1 = B[K1 - 1, K2 - 1]
S6     B[K1, K2] = A[K1 - 1, K2]

L3   for K1 = 1, ub1
L4    for K2 = 1, ub2
S4     C[K1, 2K2] = D[K1, 2K2]
S7     D[K1 - 1, 2K2] = C[K1 - 1, 2K2]

/** Final Partitioning = {(S1, S5), (S2, S3, S6), (S4, S7)} **/
```

Figure 6.14. Sample code: after application of data locality algorithm.

```
L1   for I1 = ...
S1    ARRAY[I1] = ...
L2   for J1 = ...
S2    ... = ARRAY[J1 - 1]
```

Figure 6.15(a). Sample code: parallel loops, but no data locality.

```
L1   for I1 = ...
S1    ARRAY[I1] = ...
S2    ... = ARRAY[I1 - 1]
```

Figure 6.15(b). Sample code: data locality, but no parallel loops.

the lesser weight. That is, if parallelism has a higher weight, then the example source code is transformed as shown in Figure 6.15(a). If data locality has a higher weight, then the source code is transformed as shown in Figure 6.15(b).

6.6 Imperfect-only-child loop nests on vector architecture machines using the extended transformation set

Vectorization is the process of modifying source code for execution on a vector machine. The legality criteria for vectorization was given in Section 2.4. However, the legality criteria defines what must be true for vectorization to occur, but does not describe what vectoriza-

tion *is*. Vectorization is *SIMD* execution of assignment statements, that is, all iterations of an assignment statement S_i execute before any iterations of assignment statement S_j begin execution, where $i < j$.

As an example, the source code in Figure 6.16(a) is shown before vectorization. The source code after vectorization is shown in Figure 6.16(b) and uses typical vector notation. The notation states that all m iterations of S_1 execute simultaneously. After completion of S_1, then all m iterations of S_2 execute simultaneously. The loop L_1 is implied in the array notation and is eliminated from the source code shown in Figure 6.16(b).

```
L1  for I1 = 1,m
S1    A[I1] = B[I1]
S2    D[I1] = A[I1]
```

Figure 6.16(a). Sample code: before vectorization

```
S1  A[1 : m] = B[1 : m]
S2  D[1 : m] = A[1 : m]
```

Figure 6.16(b). Sample code: after vectorization.

Some objectives of a compiler for a vector machine are to:

- minimize the work done in each loop nest;
- move the legal vectorizable loop with the greatest iteration count to the innermost position;
- match vector register length to iteration count for the innermost loop;
- evaluate if vectorization is worthwhile; and
- ensure that subscript expressions remain simple.

Each of these objectives is described in further detail below.

A compiler for a vector machine uses loop fission to minimize the work done in each loop nest. Vectorization of a loop L_i in a loop nest is applied universally to all statements $(S_1, \ldots, S_m)$ in the loop nest. If any statement(s) in the nest are non-vectorizable, then *none* of the statements are vectorized. Loop fission is used to partition the statements into the smallest, integral sets, namely Π-blocks. Separation into Π-blocks isolates the non-

vectorizable partitions of assignment statements so that the remaining partitions can be vectorized.

One objective of a vector machine is to execute as many iterations as possible as vector operations. We assume that vector operations are applied to *only* the innermost loop of a loop nest. Correspondingly, we want to permute the loop with the greatest iteration count L_{max} to the innermost position. As with previous algorithms, the legality condition of loop permutation does not allow the transformation to be applied across imperfectly nested loops. Loop L_{max} is found in the innermost region of the loop nest. As an example, the source code shown in Figure 6.18 has assignment statement S_1 nested 1-deep and assignment statements S_2 through S_6 nested 3-deep. Loop L_{max} is either of loops L_2 or L_3, but cannot be loop L_1 since loop L_1 cannot be permuted innermost.

The cost of loading a vector register and executing vector operations may exceed the value of the resulting speedup in vector mode. As an example, if the vector register length is 256 elements, but the iteration count of loop L_{max} is 32, then vector operations may not be justified. That is, it may be more efficient to execute loop L_{max} in sequential mode. Therefore, a machine-dependent *threshold* is specified. If the iteration count of loop L_{max} is below the threshold value, then the assignment statements are executed sequentially. If the iteration count of loop L_{max} is above the threshold value, then the assignment statements are executed as vector operations.

Assuming that the iteration count of loop L_{max} exceeds the threshold value, it is strip mined according to some characteristic length of the target machine, such as vector register length. As examples, the Convex C100 and C200 series vector machines have 128 element vector registers [29]. Strip mining ensures that the innermost loop is optimized so that the iteration count of the innermost loop and the vector register length exactly match.

Finally, subscript expressions need to remain simple for arrays in a loop nest that is vectorized. Typically, a *gather-scatter* directive consolidates a sparse array for vectorization, then redistributes it after vector operations. Optimization manuals for vector machines imply that gather-scatter only works for simple subscript expressions, so we assume that all subscript expressions are linear functions of one index variable plus a constant. This as-

sumption eliminates the possibility of using a wavefront transformation because it typically results in complex, coupled subscript expressions.

A newly developed algorithm for optimizing source code for a vector machine, named **ext_vector**, uses the extended and kernel set of loop transformations is given in Figure 6.17. Specifically, the algorithm uses loop normalization, fission, permutation, and blocking to obtain vectorization.

As an example, Algorithm **ext_vector** is applied to the source code shown in Figure 6.18. The source code has an imperfect-only-child loop nest because assignment statement S_1 is nested between loops L_1 and L_2. Also, loops L_1 through L_3 are initially normal, which greatly simplifies the mathematics of the following discussion without compromising the generality of Algorithm **ext_vector**.

Step 1. Normalize all loops. As mentioned, the source code shown in Figure 6.18 already has normal loops.

Step 2. Find the strongly connected components (SCC's) of the loop nest. SCC's of the dependence graph are sets of assignment statements that are in cycles, and individual assignment statements. Thus, the SCC's for the source code are $\{(S_1),(S_2,S_3),(S_4),(S_5),(S_6)\}$. The dependence from assignment statement S_5 to S_6 was initially loop-carried, but after loop fission is loop-independent. The distance vector (0,0,1) is eliminated from the distance vector set and the new distance vector set $D^{(1)}$ is $\{(0,1,2),(1,0,0)\}$.

Step 3(A). For each Π-block, find the permutable loop with the highest iteration count. There are two unique depths at which assignment statements are nested ($q = 2$) and so $depth[1 \ldots 2] = [1,3]$. That is, assignment statement S_1 is nested 1-deep and assignment statements S_2 through S_6 are nested 3-deep. Loop L_{max} is chosen between loop $L_{depth[q-1]} = L_{depth[1]} = L_1$ and $L_{depth[q]} = L_{depth[2]} = L_3$. Loop L_2 has an iteration count of 10000, and loop L_3 has an iteration count of 32. Thus, loop L_{max} is assigned loop L_2.

Algorithm ext_vector

Optimizes source code for execution on a vector architecture machine using the kernel and extended sets of loop transformations. Specifically, the objectives of the algorithm are to: minimize the work done in each loop (loop fission); move the vectorizable loop with the greatest iteration count to the innermost position (loop permutation); match the vector length to the iteration count of the innermost loop (loop blocking); evaluate if vectorization is worthwhile; and ensure that the subscript expressions remains simple.

Input:

$\{L_1, \ldots, L_n\}$ /***an n-deep loop nest***/
$G(V, E)$ /***a dependency graph for the source code where V are the assignment statements and E are the dependences in the source code***/
$\{S_1, \ldots, S_m\}$ /***set of assignment statements in the loop nest***/
b_i /***blocking factor, such as vector register length***/
$depth[1 \ldots q]$ /***ordered set of unique depths at which assignment statements are nested***/

Output:

$\{\Psi_1, \ldots, \Psi_q\}$ /***sets of assignment statements optimized for vector machines***/

Procedure:

1. /***Normalize all loops in the loop nest.***/
   For all L_i, $1 \leq i \leq n$, **normalize**(L_i)

2. /***Find the strongly connected components (Π-blocks) for the loop nest.***/
   $\{\Psi_1, \ldots, \Psi_q\}$ = SCC(G)

3. /***For each Π-block:***/
   For all Ψ_j, $1 \leq j \leq q$
   (A) /***Find the permutable loop with the highest iteration count L_{max}.***/
   L_{max} = HIGHEST_IT_COUNT(Ψ_j) between $L_{depth[q-1]}$ and $L_{depth[q]}$

   (B) /***If the iteration count of L_{max} is less than threshold, where threshold = b_i***/
   threshold = b_i
   If L_{max} < *threshold*
   Then terminate.

   (C) Else
   (i)/***Move L_{max} innermost***/
   permute(L_{max}, L_n)
   (ii)/***Block innermost loop by vector register length.***/
   block(L_n, b_i)
   (iii)/***Put blocked loop in vector notation***/
   vectorize(L_{n+1})

Figure 6.17. Algorithm to optimize source code for a vector machine using the extended transformations on imperfect-only-child loop nests.

```
L1   for I1 = 1,3
S1    A[I1] = ...
L2    for I2 = 1,10000
L3     for I3 = 1,32
S2      B[I1,I2,I3] = C[I1 - 1,I2,I3]
S3      C[I1,I2,I3] = B[I1,I2 - 2,I3 - 1]
S4      E[I1 + 1,I2,I3 - 1] = ...
S5      F[I1,I2,I3] = ...
S6      F[I1,I2,I3 - 1] = ...
/**D = {(0,2,1),(1,0,0),(0,0,1)}
bi = 256
SCC = {(S1),(S2,S3),(S4),(S5),(S6)}**/
```

Figure 6.18. Sample code: before application of Algorithm **ext_vector**.

Step 3(B). If the loop L_{max} has a low iteration count, then terminate. Else, we continue. The iteration count of L_{max} is 10000. The threshold is equal to the block size b_i, and has the value of 256 in this example. Therefore, we continue.

Step 3(C(i)). Interchange loop L_{max} with the innermost loop. The second loop L_{max} and the third loop L_3 are interchanged. The transformation matrix representing the interchange is:

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

and the transformed distance vector set is:

$$D^{(2)} = \{(0,1,2),(1,0,0)\}.$$

Step 3(C(ii)). Block the innermost loop by $b_i = 256$. The innermost loop is blocked so that the iteration count matches the vector register length. The transformation matrix is:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & b_o & 0 \\ 0 & b_i & 0 \end{bmatrix}$$

The transformed distance vector set is:

$$\begin{aligned} D^{(3)} &= \{(0,1,\lfloor\tfrac{2}{256}\rfloor,2),(0,1,\lceil\tfrac{2}{256}\rceil,2),(1,0,\lfloor\tfrac{0}{256}\rfloor,0),(1,0,\lceil\tfrac{0}{256}\rceil,0)\} \\ &= \{(0,1,0,2),(0,1,1,2),(1,0,0,0)\}. \end{aligned}$$

Step 3(C(iii)). Place the source code in vector notation. In the transformed distance vector set $D^{(3)}$, all of the loop-carried dependences are carried at a depth less than four. Thus, no dependence cycles exist for the innermost blocked loop and the source code can be vectorized. The source code shown in Figure 6.19(a) shows transformed source code before being placed in vector notation, and Figure 6.19(b) shows the same source code in vector notation.

```
L1   for I1 = 1,3
S1    A[I1] = ...
L1   for I1 = 1,3
L3    for I3 = 1,32
L2'    for I2' = 1,10000,bi
L2      for I2 = I2',min(I2' + bi - 1,10000)
S2       B[I1,I2,I3] = C[I1 - 1,I2,I3]
S3       C[I1,I2,I3] = B[I1,I2 - 2,I3 - 1]
L1   for I1 = 1,3
L3    for I3 = 1,32
L2'    for I2' = 1,10000,bi
L2      for I2 = I2',min(I2' + bi - 1,10000)
S4       E[I1 + 1,I2,I3 - 1] = ...
L1   for I1 = 1,3
L3    for I3 = 1,32
L2'    for I2' = 1,10000,bi
L2      for I2 = I2',min(I2' + bi - 1,10000)
S5       F[I1,I2,I3] = ...
L1   for I1 = 1,3
L3    for I3 = 1,32
L2'    for I2' = 1,10000,bi
L2      for I2 = I2',min(I2' + bi - 1,10000)
S6       F[I1,I2,I3 - 1] = ...
```

Figure 6.19(a). Sample code: ready for vectorization.

```
S1   A[1 : 3] = ...
L1   for I1 = 1,3
L3    for I3 = 1,32
L2'    for I2' = 1,10000,bi
S2       B[I1,I2' : I2' + bi - 1,I3] =
                C[I1 - 1,I2' : I2' + bi - 1,I3]
S3       C[I1,I2' : I2' + bi - 1,I3] =
                B[I1,I2' - 2 : I2' + bi - 3,I3 - 1]
L1   for I1 = 1,3
L3    for I3 = 1,32
L2'    for I2' = 1,10000,bi
S4       E[I1 + 1,I2' : I2' + bi - 1,I3 - 1] = ...
L1   for I1 = 1,3
L3    for I3 = 1,32
L2'    for I2' = 1,10000,bi
S5       F[I1,I2' : I2' + bi - 1,I3] = ...
L1   for I1 = 1,3
L3    for I3 = 1,32
L2'    for I2' = 1,10000,bi
S6       F[I1,I2' : I2' + bi - 1,I3 - 1] = ...
```

Figure 6.19(b). Sample code: in vector notation.

6.7 Imperfect-only-child loop nests on a shared-memory machine using the extended transformation set

Our model of a shared-memory machine is *MIMD*. If p is the number of processors, then each processor $P_i, 1 \leq i \leq p$, may operate on different instructions and a different data stream. As an example, the source code in Figure 6.20(a) is shown before execution on a *MIMD* shared-memory machine. Figure 6.20(b) shows the execution of the assignment statements S_1 and S_2 on processors P_1 through P_m. (For simplicity, we assume that the number of processors and number of iterations are the same, that is, $p = m$.) Note that there is no implicit or explicit barrier synchronization between execution of S_1 and S_2 on processors P_i and P_j, where $i \neq j$. That is, assignment statement S_2 may finish execution on processor P_i before processor P_j begins execution of statement S_1.

L_1 **for** $I_1 = 1, m$
S_1 A[I_1] = B[I_1]
S_2 D[I_1] = A[I_1]

Figure 6.20(a). Sample code: before parallelization

	P_1	$P_2 \ldots$	P_m
S_1	A(1)=B(1)	A(2)=B(2)...	A(m)=B(m)
S_2	D(1)=A(1)	D(2)=A(2)...	D(m)=A(m)

Figure 6.20(b). Statement distribution on a SMS.

The objectives of a compiler for a shared-memory machine differ significantly from those of a vector architecture machine. Specifically, the objectives for a shared memory machine are to:

- maximize the work done in each loop nest;
- move parallel loops outermost;
- maximize iteration count of the parallelized loops;

In vector machines, the objective is to minimize the work done in each loop, because the majority of the overhead on a vector machine is loading the vector registers. In contrast, significant overhead for a shared-memory machine is physically distributing the work to

the available processors. Therefore, a primary objective of a compiler for a shared-memory system is to maximize the work done in each loop, which minimizes the work distribution overhead. Loop fusion is used to combine assignment statements from adjacent loop nests, which maximizes the work done in each loop nest.

Algorithm **mod_kernel_fine** (Figure 6.4) is a modified wavefront transformation that applies to imperfect-only-child loop nests. The algorithm for shared-memory machines, named **ext_sms** and shown in Figure 6.21, uses Algorithm **mod_kernel_fine** to obtain the maximal amount of parallelism.

One of the objectives for a shared-memory machine is to move the parallelizable loops outermost. As in previous algorithms, we cannot violate the legality condition for loop permutation. That is, if the outermost assignment statement S_1 is nested j-deep, then the two loops with the highest iteration count are found between loops L_1 and L_j and moved outermost. The two loops with the highest iteration count are named L_{max1} and L_{max2}, respectively. In previous algorithms, only the single loop with the highest iteration count was found. In this algorithm, we find two loops L_{max1} and L_{max2} because the final stage of the Algorithm **ext_sms** is scheduling.

As mentioned earlier, work distribution on a shared-memory machine incurs significant overhead. The actual work distribution protocol is referred to as *scheduling*. There are two types of scheduling: *self-scheduling*, in which the work distribution is determined at run-time; and *pre-scheduling*, in which the work distribution is determined at compile time. Loop collapsing aids scheduling by combining parallel loops that are distributed to processors. That is, loop collapsing decreases the amount of overhead associated with work distribution by simplifying the loop structure to which scheduling protocols are applied. Therefore, we collapse the two loops with the highest iteration counts (L_2 and L_3) in order to aid scheduling.

An algorithm that optimizes source code for a shared-memory machine, named **ext_sms**, uses the extended and kernel set of loop transformations and is given in Figure 6.21. The algorithm uses loop normalization, fusion, skewing, permutation, and collapsing.

Algorithm ext_sms

Optimizes source code for execution on a shared-memory machine using the kernel and extended sets of loop transformations. Specifically, the objectives of the algorithm are to: maximize the work done in each loop (loop fusion); move the parallelizable loops with the greatest iteration count to the outermost position (loop permutation); and maximize the iteration count of the innermost loops (loop collapsing).

Input:

$\Theta_1 = \{M_1, \ldots, M_n\}$ /***an n-deep loop nest***/
$\Theta_2 = \{N_1, \ldots, N_n\}$ /***another n-deep loop nest***/
depth1$[1 \ldots q_1]$ /***ordered set of unique depths at which assignment statements in* Θ_1 *are nested***/
depth2$[1 \ldots q_2]$ /***ordered set of unique depths at which assignment statements in* Θ_2 *are nested***/

Output:

L /***the transformed loop nest optimized for a SMS***/

Procedure:

1. /***Normalize all loops in both loop nests.***/
   For all i, $1 \leq i \leq n$
   normalize(M_i)
   normalize(N_i)

2. /***Fuse loop nests to maximize work.***/
   $L =$ **fuse**(Θ_1, Θ_2)
   depth = *depth1* $\cup$ *depth2*

3. /***Apply wavefront to obtain* $n - q$ *degrees of parallelism:***/
   mod_kernel_fine(L)

4. /***Find parallel loops in nest with highest iteration count and move to positions* L_2 *and* L_3.**/
   $L_{max1} =$ **max**$(L_2, \ldots, L_{depth[1]})$
   $L_{max2} =$ **max**$(L_2, \ldots, L_{depth[1]} - L_{max1})$
   permute(L_2, L_{max1})
   permute(L_3, L_{max2})

5. /***Collapse loops* L_2 *and* L_3 *to aid scheduling.***/
   collapse(L_2, L_3)

Figure 6.21. Algorithm to optimize source code for a shared memory machine using the extended transformations on imperfect-only-child loop nests.

Algorithm **ext_sms** is applied to the source code shown in Figure 6.22. The first loop nest Θ_1 has four loops M_1 through M_4, and assignment statements are nested at two unique depths. That is, $q_1 = 2$ and $depth1[1 \ldots 2] = [3, 4]$. The second loop nest Θ_2 also has four loops N_1 through N_4. All assignment statements have perfect nesting. Therefore, $q_2 = 1$ and $depth2[1] = [4]$. For simplicity, all loops in both the first code block Θ_1 and the second code block Θ_2 are normal.

```
M1   for I1 = 1,1000
M2    for I2 = 1,32
M3     for I3 = 1,500
S1      A[I1,I2,I3] = ...
M4     for I4 = 1,50
S2      B[I1,I2,I3,I4] = ...

N1   for J1 = 1,1000
N2    for J2 = 1,32
N3     for J3 = 1,500
N4     for J4 = 1,50
S3          ... = B[J1,J2,J3,J4 - 2]
```

Figure 6.22. Sample code: before application of Algorithm **ext_sms**.

Step 1. Normalize all loops. As mentioned, the source code shown in Figure 6.22 already has normal loops.

Step 2. Fuse Θ_1 and Θ_2. After loop fusion, the assignment statements in loop nest Θ_1 and Θ_2 are combined into one loop nest, named L. The loop-independent dependence between statement S_2 and S_3 becomes loop-carried, and the dependence vector set $D^{(2)}$ is $\{0, 0, 0, 2)\}$. Assignment statement S_1 remains nested 3-deep in the fused source code.

Step 3. Apply modified wavefront transformation. Algorithm **mod_kernel_fine** is applied to the loop nest to obtain maximal parallelism. The imperfectly nested assignment statement S_1 causes wavefront transformation to be applied to loops L_1 through L_3 and to loop L_4. Loop L_3 is skewed by loop L_1 and L_2, then loop L_3 is moved outermost. The resulting dependence vector is $D^{(3)} = \{(0, 0, 0, 2)\}$. According to the parallelism criteria (Expression (6.1)), loops L_1 through L_3 can be parallelized.

Step 4. Find two permutable loops with the highest iteration counts and move them outermost. The iteration count of loops L_1, L_2, and L_3 are 1000, 32, and 500, respectively. Therefore, loop L_1 has the highest iteration count ($L_{max1} = L_1$) and loop L_3 has the second highest iteration count ($L_{max2} = L_3$). Loops L_{max1} and L_{max2} are moved outermost using loop permutation. The distance vector set remains unchanged.

Step 5. Collapse the outermost loops. Finally, the outermost two loops are collapsed to aid scheduling. The resulting source code is shown in Figure 6.23. Note that the two outermost loops are parallel and the transformed distance vector set is $D^{(5)} = \{((0 \times 500) + 0, 0, 2)\} = \{(0, 0, 2)\}$.

```
L1,2  forall K1,3 = 1, 500000
L3     forall K2 = 1, 32
         if (K1,3 mod 500) == 0
          then K1 = K1,3/500
          else K1 = ⌈K1,3/500⌉ + 1
         K3 = ((K1,3 − 1) mod 500) + 1
S1       A[K1, K2, K3] = ...
L4       for K4 = 1, 50
S2        B[K1, K2, K3, K4] = ...
S3              ... = B[K1, K2, K3, K4 − 2]
```

Figure 6.23. Sample code: after application of Algorithm **ext_sms**.

CHAPTER 7

Implementation Model

This chapter gives the *requirements analysis* for the development of an environment that supports the theoretic model presented in previous chapters. The approach used for system analysis is the *Object Modeling Technique* (**OMT**) [30], which uses three diagramming notations to model orthogonal viewpoints of the system. The *Object Model* describes system structure and the relationships of objects to each other. The *Dynamic Model* describes the dynamic behavior of the system as it changes from state to state. Finally, the *Functional Model* captures data flow through the system. The three models describe complementary aspects of the system and can be used to guide the entire development process.

7.1 Overview

The implementation prototype is targeted for a graphics-based framework for matrix-based loop transformations to target source code. The loop transformations: loop permutation, loop reversal, loop skewing, loop normalization, loop blocking, strip mining, cycle shrinking, loop collapsing, loop coalescing, loop fission, and loop fusion, will be represented by matrices.

Transformation matrices are determined based upon machine-independent goals, such as parallelism or data locality; or machine-dependent goals, such as optimization for vector or shared memory systems. Finally, the transformations can be user-directed, where the user defines the transformation sequence and the matrix representing the sequence.

7.1.1 User Scenario

The following is a typical scenario of a session with the proposed framework.

1. The user selects a source code file to process.
2. The system checks the syntax of the source code.
3. The system loads the source code file into a window.
4. The user selects a portion of source code (e.g. loop nest, block, all) upon which to run dependence analysis.
5. The user selects which dependence test to use, such as:
   - Banerjee Test [15, 31];
   - GCD Test [15];
   - Lambda Test;
   - Power Test [32, 33];
   - Omega Test [34].
6. The system runs the dependence test on the target source code and determines the distance vector set D, which is displayed to the user in a window.
7. The system allows the user to specify either machine-directed, goal-directed, or user-directed transformation matrix.
   (a) If machine-directed is chosen, then the system prompts the user for machine type.
      - If vector machine is chosen, then the system generates a transformation sequence based upon Algorithm **ext_vector** for vector machines.
      - If shared-memory machine is chosen, then the system generates a transformation sequence based upon Algorithm **ext_sms** for shared-memory machines.

   (b) If goal-directed is chosen, then the system prompts the user for a goal.
      - If parallelism is chosen, then the system generates a transformation matrix T based upon Algorithm **ext_coarse**.
      - If data locality is chosen, then the system generates a transformation matrix T based upon Algorithm **ext_locality**.

   (c) If user-directed is chosen, then the user chooses a transformation from a menu and a matrix-based representation of the transformation sequence is displayed in a window. If the transformation that the user chooses is illegal, then a corresponding error message is returned to the user.
      - If permutation is chosen, then the user is queried for two loops.
      - If reversal is chosen, then the user is asked which loop.
      - If skewing is chosen, then the user is asked to supply the outer loop, inner loop, and the value of the skewing factor.
      - If normalization is chosen, then the user is asked which loop.

- If fission is chosen, then the user is asked to partition assignment statements into two Ψ-blocks by choosing individual or blocks of assignment statements.
- If fusion is chosen, then the user chooses two adjacent Θ-blocks.
- If blocking is chosen, then the user is queried for the loop and the blocking factor.
- If collapsing is chosen, then the user is asked which two adjacent loops.

8. The transformation matrix T is displayed in a window, and the distance vector set D' is displayed in another window.
9. The user can request at any time to check the global postcondition by evaluating D'.
10. The user can request at any time to have the source code transformed based upon the current T.
11. If the user requests to transform the source code, then the system checks the global postcondition. If the global postcondition is *true*, then the code is updated. If it is not, then an error message is given, and the user is prompted to apply more transformations.
12. Updated code is displayed in a separate window.
13. The user may save the transformed source code and distance vector set to a file for future use.

Figure 7.1 shows a sample graphical user interface for the system. The source code transformation process proceeds from left to right using this GUI. Popup menus give the user choices of dependence tests, transformation objectives, and source code mapping.

7.2 Requirements

Unimodular matrices are a method of representing a sequence of kernel transformations (loop permutation, reversal, and skewing) in matrix form. The dependences in the source code are represented by the dependence vector set D, and the transformation sequence is represented by a matrix T.

The distances in the initial distance vector set D are all lexicographically positive (*global precondition*). After transformation, the transformed distance vectors in the distance vector set must be lexicographically positive (*global postcondition*). Additional source code transformation goals, such as parallelism, are evaluated based upon the form of the transformed distance vector set.

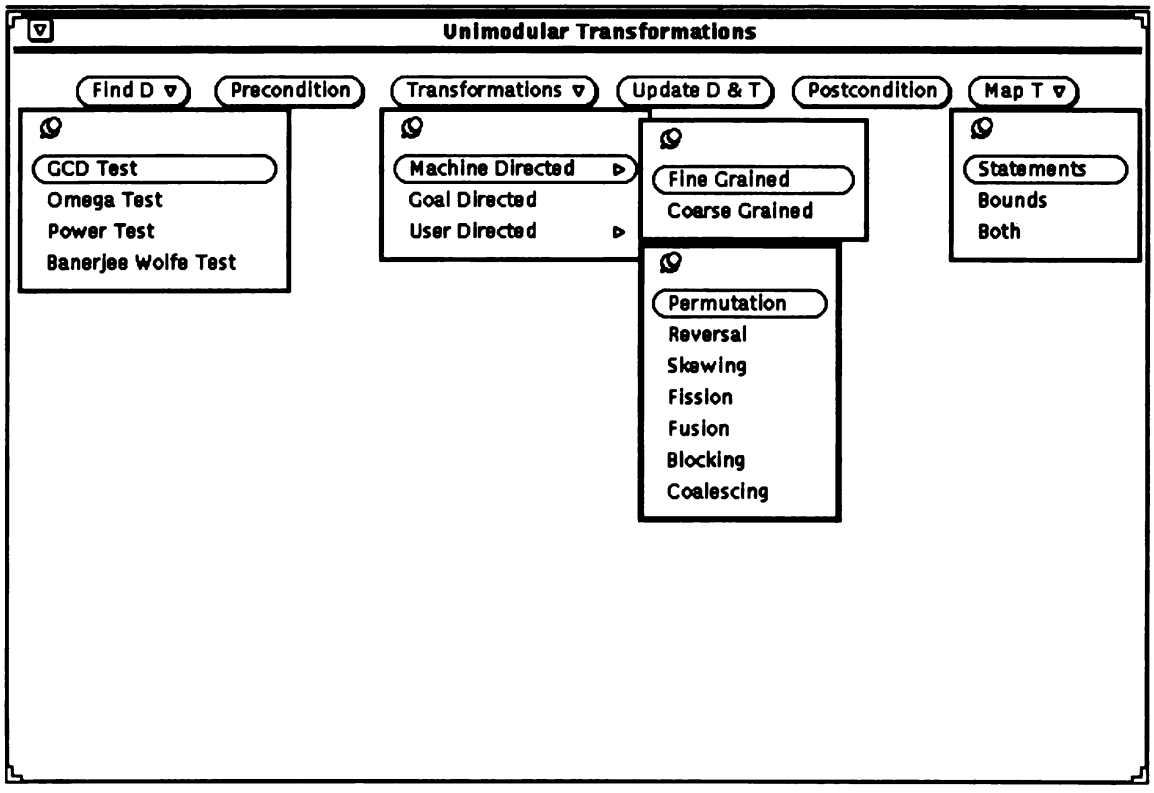

Figure 7.1. Graphical user interface.

The transformations that are represented include the kernel unimodular transformation set (loop permutation, reversal, and skewing), as well as the extended transformation set (loop normalization, blocking, collapsing, fission, and fusion). A sequence of kernel and extended transformations is represented by the product of the matrices representing the individual transformations (*composability*). The source code is mapped only once after the final transformation matrix T is determined.

The transformation matrix T is determined in one of three ways: 1) machine-dependent: the system generates T based upon machine characteristics; 2) machine-independent: the system generates T based upon machine-independent goals, such as parallelism and data locality; or 3) user-directed: T is generated based upon user directives.

The legality of the individual transformations is determined based upon the preconditions discussed in Chapters 3 through 5. If an individual transformation is illegal, then a popup window displays the legality criteria violation. The system applies transformations to both perfect and imperfect loop nests. The transformation process continues, regardless of the source code structure, as long as the individual transformations are legal.

7.3 OMT Analysis

This section contains the analysis of the system using the Object Modeling Technique (**OMT**). The three submodels that comprise the model are the *object model*, *dynamic model*, and *functional model*. The diagrams that represent the respective models are the *object diagram*, *state diagram*, and *data flow diagram*.

7.3.1 Object Model

The object model describes the overall static structure of the system, where objects are discrete, distinguishable entities. The relationships between objects are represented in diagrams, where a triangle ($\triangle$) means inheritance and a diamond ($\diamond$) means aggregation. Five object models are described below: 1) system; 2) dependences; 3) transformation matrices; 4) transformation sequences; and 5) source code.

The object model for the system is given in Figure 7.2. The parallel compiler consists of a dependence test utility, such as *Tiny* [19, 35]. Also, the compiler has a GUI, a transformation matrix T builder, and a source code mapper (for example, the algorithms developed my Wolf and Lam [4]). The transformation matrix builder consists of a global precondition checker, a global postcondition checker, and algorithms. The algorithms reflect operation of the system in either machine-independent, machine-dependent, or user-directed mode. Finally, the mapper has two components: one maps the distance vector set D, and another maps the source code.

The relationships between pages of the object model are represented by associations in Figure 7.2. Dependence tests generate the distance vector set D, and also, the distance vector set D is mapped by the distance vector set D mapper. The source code is mapped by the source code mapper. Finally, the global transformation matrix is generated by the algorithms in the system.

The object model representing dependences is shown in Figure 7.3. A distance vector set D consists of distance vectors, where each distance vector consists of distances. The distance vector set D can be represented by a dependence graph, where each node is an assignment statement and there is a one-to-one correspondence between arcs in the dependence graph and distance vectors.

Figure 7.4 shows the object model for the transformation matrices, and also, the association between the transformation matrix and the transformation sequence (Figure 7.5). Each global matrix is a product, or sequence, of transformations. Each transformation is comprised of unimodular and elementary matrices. A unimodular matrix is a type of elementary matrix with special properties, which represents loop permutation, reversal, and skewing.

The object model for the transformation sequence is shown in Figure 7.5. A transformation sequence is comprised of an ordered sequence of transformations. Each transformation is either a kernel unimodular transformation (loop permutation, reversal, and skewing) or an extended transformation (loop normalization, blocking, collapsing, fission, and fusion).

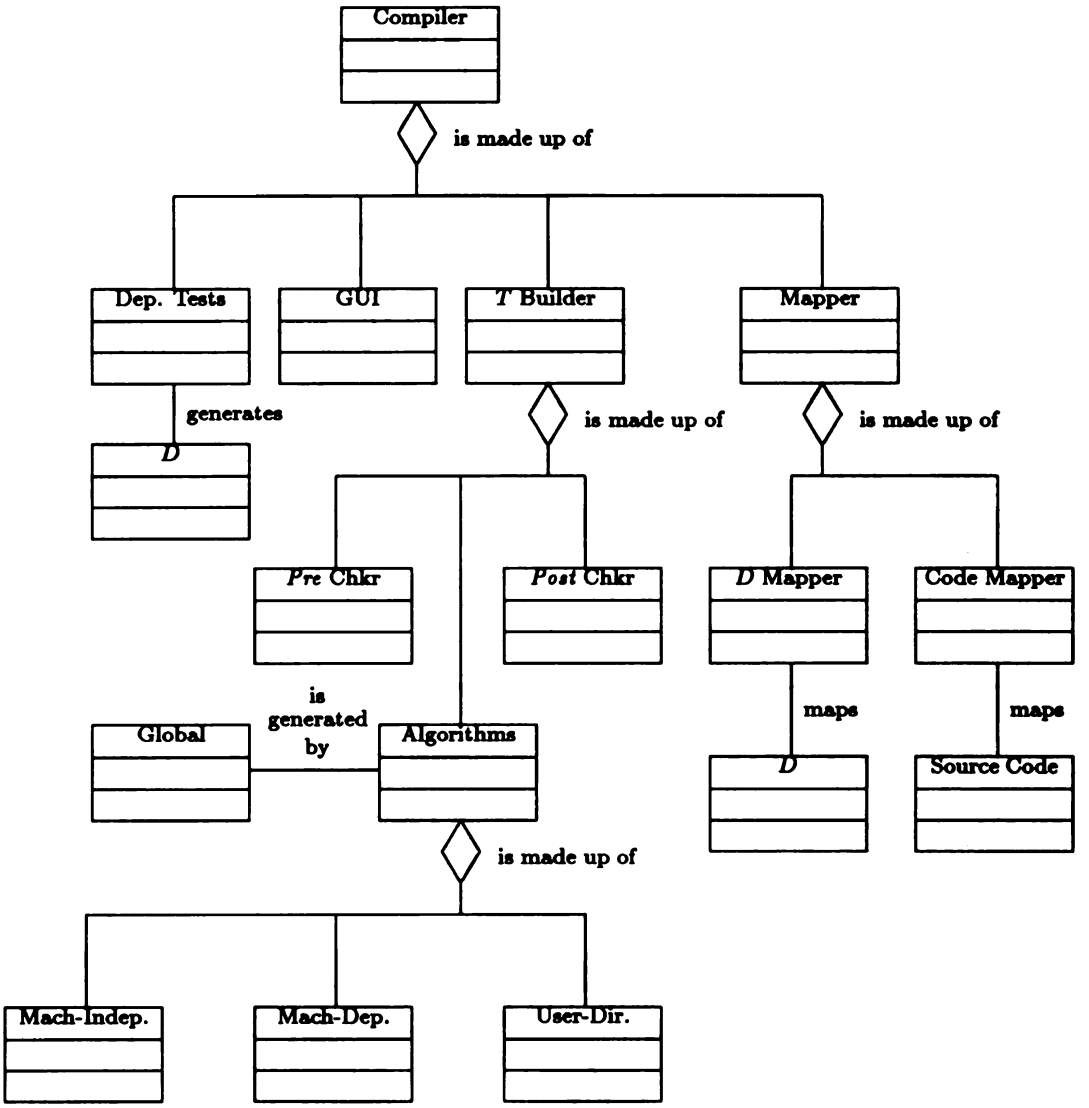

Figure 7.2. Object diagram for overall system.

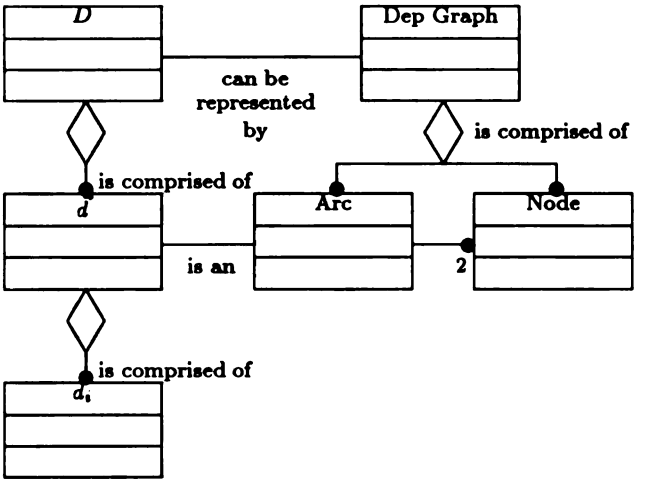

Figure 7.3. Object diagram for source code dependences.

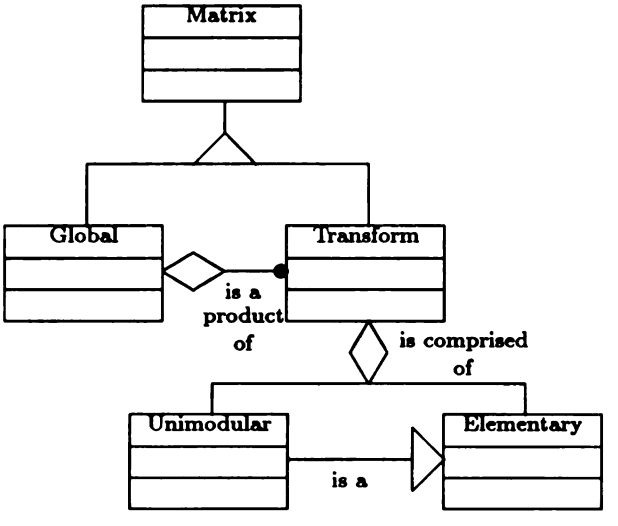

Figure 7.4. Object diagram for transformation matrices.

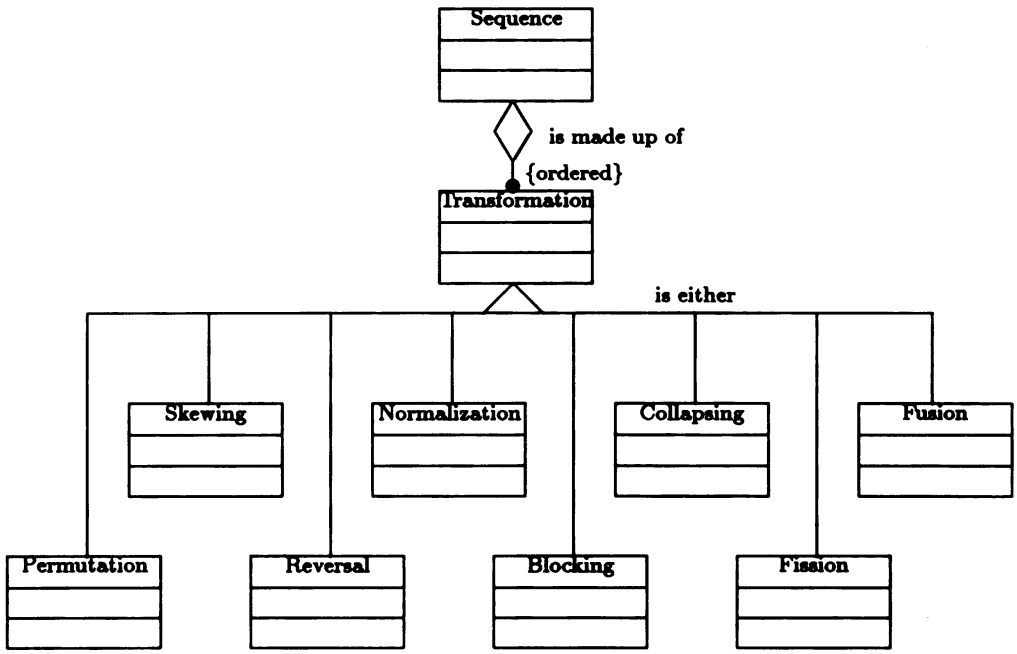

Figure 7.5. Object diagram for transformation sequence.

Figure 7.6 shows the object model for source code, which is the target to which the transformations are applied. Because of the recursive structure of the source code (i.e., a loop may be comprised of other loops), the object model may not be the best representation of the source code. However, the object model is used here for consistency with the **OMT** modeling approach.

In Figure 7.6, source code is comprised of loop nests, where each loop nest consists of loops. Each loop has a header (i.e., `for` $I_i = lb_i, ub_i, s_i$) which has an index variable, lower bound, upper bound, and step; assignment statements; and a tail (i.e., `endfor`). Assignment statements are comprised of *use* variables and *def* variables, which are typically arrays with subscript expressions. Θ-blocks are defined in terms of loop nests and contain groupings of assignment statements. A Ψ-block is a specific type of Θ-block with special dependence characteristics. Finally, a Π-block is a strongly connected component of the dependence graph, and Ψ-blocks are comprised of Π-blocks.

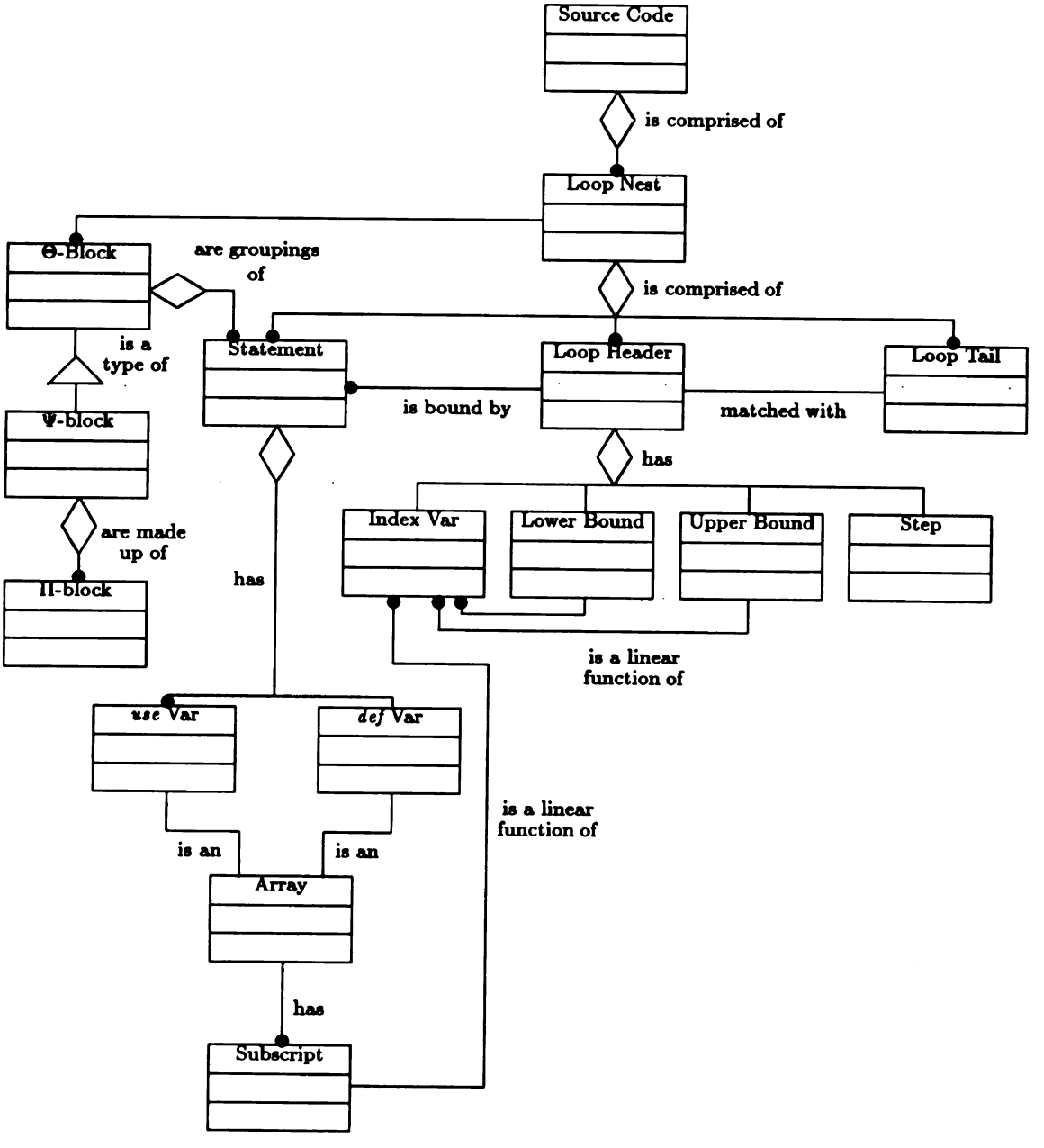

Figure 7.6. Object diagram for source code.

A more rigorous representation of source code is to use *Backus-Naur Form* (*BNF*). The *BNF* of a target language fully describes its structure in a succinct manner. The *BNF* for a modified imperative language is shown in Figure 7.7.

7.3.2 Dynamic Model

The dynamic model describes the behavior of the system over time. State diagrams represent the dynamic model, where nodes in the diagram are states and arcs are transitions (events) between states. In the state diagram, "ur" is shorthand for "user request". Also, the parentheses following some transitions are conditions that must be true in order for the transition to trigger.

The state diagram for the system is shown in Figure 7.8. The system is idle until the user requests that a file be loaded. If the syntax of the file is incorrect, then the system enters an error state until a file with good syntax is entered. Once a file with good syntax is loaded, the user can request a dependence test to calculate the initial distance vector set $D^{(0)}$.

There are three possible paths that the user chooses after the distance vector set is calculated. The machine-independent and machine-dependent paths generate the transformation matrix T using the algorithms discussed in Chapter 6. The third alternative is for the user to generate her/his own transformation sequence.

There is extensive error checking while the user is choosing individual transformations. Each time a loop transformation choice is made, the legality of the transformation is evaluated. If the transformation is illegal, then the user is allowed to undo a transformation and choose another transformation. This process continues until a legal transformation is chosen. Once the user is satisfied with the transformation sequence, the system checks if the global postcondition is true. If it is not true, then the user continues to choose transformations. If it is true, then the system proceeds.

All three paths converge after a transformation matrix T is determined. The user may request that the source code be mapped from initial to final form given the transformation matrix.

```
program          ::= stlist

stlist           ::= statement[';'statement]...

statement        ::= integerdecl
                 ::= realdecl
                 ::= constdecl
                 ::= assignment
                 ::= loop
                 ::= if

constdecl        ::= 'const' constassignment [','constassignment]...

constassignment  ::= ID '=' expression

integerdecl      ::= 'integer' vardecllist

realdecl         ::= 'real' vardecllist

vardecllist      ::= vardecl[','vardecl]...

vardecl          ::= ID
                 ::= ID '('[expression ':'] expression
                      [','[expression ':'] expression]... ')'

assignment       ::= lhs '=' expression

lhs              ::= ID
                 ::= ID '(' expression [',' expression]... ')'

loop             ::= ['for' | 'doall']
                      ID '=' expression ['to' | ',' | ':'] expression
                      [['by' | ',' | ':'] expression]
                      'do' stlist 'endfor'

expression       ::= ID
                 ::= ID '(' expression [',' expression]... ')'
                 ::= INTCONST
                 ::= FLOATCONST
                 ::= expression '+' expression
                 ::= expression '-' expression
                 ::= expression '*' expression
                 ::= expression '/' expression
                 ::= expression '**' expression
                 ::= '-' expression
                 ::= '+' expression
                 ::= '(' expression ')'
                 ::= expression '<' expression
                 ::= expression '<=' expression
                 ::= expression '<>' expression
                 ::= expression '>' expression
                 ::= expression '>=' expression
                 ::= expression 'mod' expression
                 ::= expression 'max' expression
                 ::= expression 'min' expression
                 ::= 'sqrt' '(' expression ')'
                 ::= 'floor' '(' expression '/' expression ')'
                 ::= 'ceiling' '(' expression '/' expression ')'
                 ::= 'max' '(' expression [',' expression]... ')'
                 ::= 'min' '(' expression [',' expression]... ')'
```

Figure 7.7. Backus-Naur Form of target imperative language (modified *Tiny* syntax).

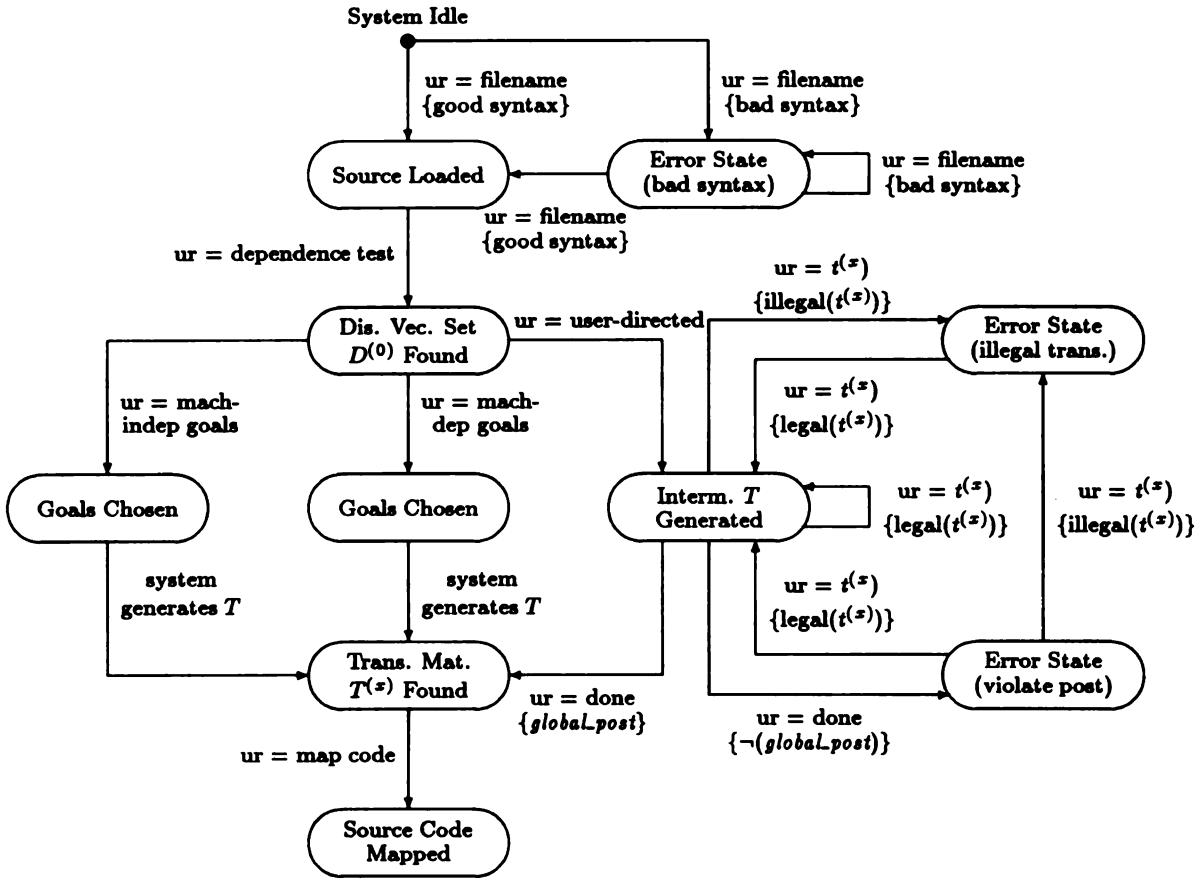

Figure 7.8. State diagram for system.

7.3.3 Functional Model

The functional model captures the flow of data through the system, and is represented by a data flow diagram (*DFD*). *DFD*'s are described in terms of levels, where each successive level shows greater detail of the data flow in the system. In a *DFD*, the ovals represent *processes*, that is, producers and consumers of data. The boxes represent *actors* that are external entities that move data through the system. Two horizontal lines represent a *data store*, and the arcs represent *data flow*.

Figure 7.9 is the level 0 data flow diagram for the implementation, also known as a context diagram. The process *Transformer* does the actual source code transformations. As input, *Transformer* needs the initial distance vector set $D^{(0)}$ and the original source code. As output, *Transformer* produces a transformed distance vector set $D^{(x)}$ and transformed source code. We assume that an external process, such as *Tiny* [19, 35], is used to determine the distance vector set from the initial source code.

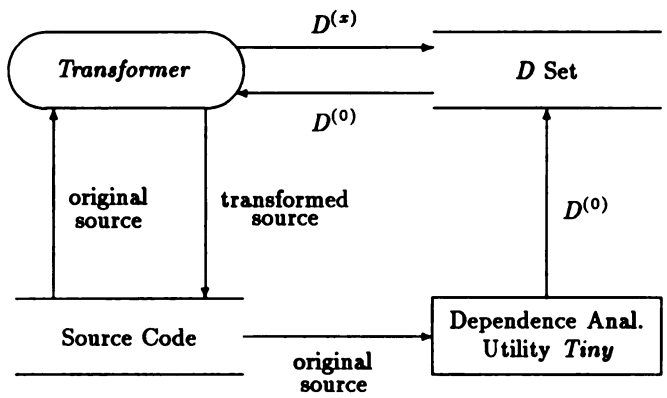

Figure 7.9. Level 0 data flow diagram.

The *Transformer* process is examined more closely in the level 1 data flow diagram shown in Figure 7.10. The initial distance vector set $D^{(0)}$ is used when determining the transformation matrix T. The source code is mapped from its initial to final form as long as both the global postcondition and goals are met, as determined from $D^{(x)} = T^{(x)} \times \vec{d} \in D^{(0)}$.

The transformation matrix and initial source code are used to map the final source code. Both the mapped source code and transformed distance vector set can be saved to external files.

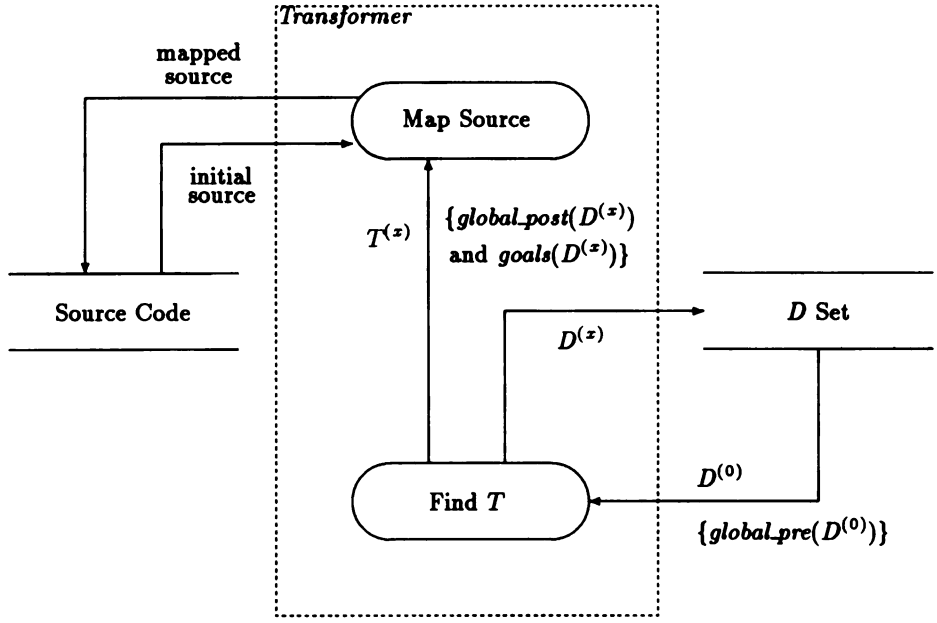

Figure 7.10. Level 1 data flow diagram.

Figure 7.11 shows the level 2 data flow diagram for finding the transformation matrix T, and assumes that the user is choosing the transformation sequence. The initial distance vector set $D^{(0)}$ is input into the process. The user chooses an elementary transformation matrix $t^{(x)}$, and if $t^{(x)}$ is legal, then a new global transformation matrix $T^{(x)}$ is calculated. After $T^{(x)}$ is calculated, the distance vector set $D^{(x)}$ is updated. The user can choose more transformations, or map the source code.

The source code is mapped according to the approach described in Section 2.7. First, inequalities, minima, and maxima are determined from the initial source code. Then the index variables are transformed based upon the transformation matrix $T^{(x)}$. Finally, new loop bounds are calculated based upon the mapped index variables and inequality set.

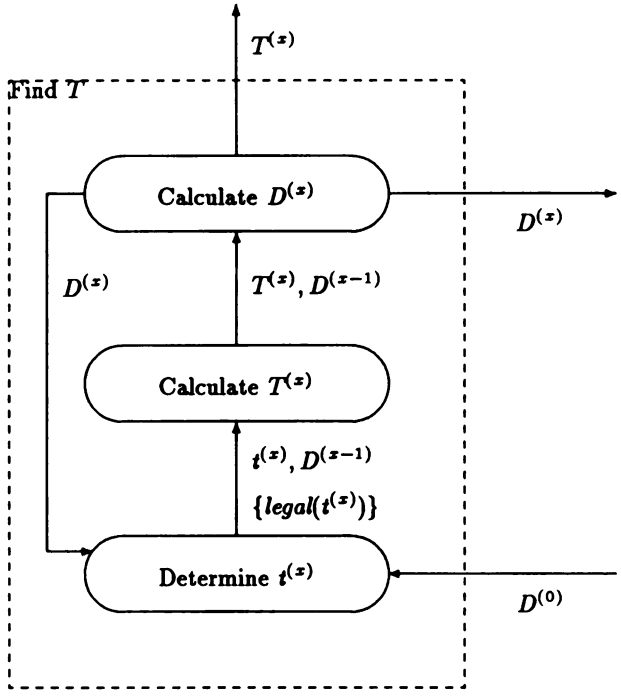

Figure 7.11. Level 2 data flow diagram for finding the transformation matrix T.

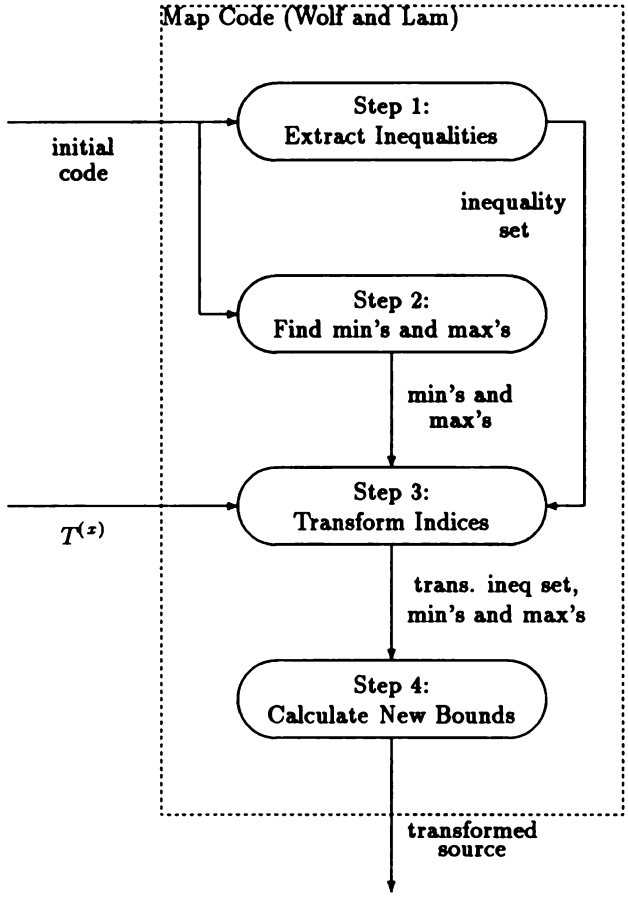

Figure 7.12. Level 2 data flow diagram for mapping source code.

CHAPTER 8

Related Work

This chapter describes current research in ordering source code transformations. Several of the approaches are matrix-based, while others are based upon formal specification of the transformations. Some of the approaches suggest static ordering of the transformations to obtain optimal transformed source code, while other approaches determine transformation ordering based upon source code structure. Finally, some representations include both scalar and loop transformations, while others contain only loop transformations.

8.1 Unimodular Matrices

Work by Banerjee [3] shows that unimodular matrices represent a sequence of transformations on a perfect doubly-nested loop nest, where all dependences are represented by distance vectors. Banerjee also describes an algorithm that *may* obtain coarse-grain parallelism and an algorithm that *always* obtains fine-grain parallelism. Wolf and Lam [4] expand upon Banerjee's work and show the use of unimodular matrices to represent a sequence of transformations on an n-nested loop, where dependences are represented as either distance or direction vectors. Wolf and Lam also give a general approach to obtain coarse-grain parallelism and prove that $n - 1$ degrees of parallelism can always be obtained from an n-nested loop.

The basis of unimodular transformations is thoroughly described in Section 2.5, and is briefly described below. Sequences of kernel unimodular transformations (loop permutation, reversal, and skewing) are represented by a transformation matrix T, where T is:

1. $n \times n$ (square);
2. all $T[i, j]$ are integers;
3. $|det\ T| = 1$.

The legality of a transformation sequence and the available parallelism are determined from the product of T and all of the distance vectors in D. Finally, an algorithm exists to map the source code from its initial to final form given the transformation matrix T.

Some restrictions to the traditional unimodular approach are that unimodular matrices only represent three kernel transformations and are only applied to perfect loop nests. These two restrictions imply that, although unimodular transformations are theoretically elegant, they have limited applicability.

Sass and Mutka [26, 28] have described two different methods for applying unimodular matrices to imperfect loop nests. The first method, called *enabling*, transforms some cases of imperfect loop nests into perfect loop nests. The second method, called the *phase method*, applies unimodular transformations directly to imperfect loop nests at the expense of increased barrier synchronization.

Sass and Mutka [28] describe a static ordering of transformations that *enables* unimodular transformations. Specifically, *scalar forward substitution* and *loop distribution* are used to transform source code into perfect loop nests. Next, unimodular transformations are used to obtain parallelism, as in the traditional approach. Finally, *loop fusion* and *common subexpression elimination* are used, as necessary, to restore the structure of the original source code. The only source code structure that cannot be transformed into perfect loop nests is source code with a transfer of control from within a loop nest, or source code with an iterative command within an alternative command.

Alternatively, Sass and Mutka [26] describe a transformation, called the *phase method*, that transforms a 2-deep, imperfect loop nest into three adjacent, 2-deep, imperfect-only-

child loop nests. The phase method has three steps: 1) calculate and lengthen the dependence vectors in the dependence vector set D; 2) invoke Banerjee's algorithm [3] to obtain fine-grain parallelism on a doubly-nested loop; and 3) map the source code into three phases with the innermost loop executing in parallel in all three phases.

8.2 Non-Singular Matrices

Ramanujam [5] and Li and Pingali [6] independently developed a similar extension of the unimodular approach using non-singular matrices. A non-singular matrix relaxes the third constraint of unimodular matrices ($|det\ T| = 1$), and allows the determinant of the transformation matrix T to be greater than or equal to one. Relaxation of this constraint allows more freedom in the choice of T, and, in fact, unimodular matrices are a subclass of non-singular matrices.

The legality of the transformation for non-singular matrices is the same as the global postcondition for unimodular matrices. That is, all transformed dependence vectors must be lexicographically positive. This condition is formally described as: $(\forall \vec{d} : \vec{d} \in D : T \times \vec{d} \succ \vec{0})$.

Also, optimal transformations for the non-singular approach are described in terms similar to optimal transformations for the unimodular approach. Specifically, fully permutable loop nests are related to *dependence cones* by Ramanujam [5]. Recall that fully permutable loop nests allow any ordering of the loops.

Using non-singular matrices, there is a method for mapping from the initial code to transformed code for both the assignment statements and the loop bounds. Although determining the assignment statements is relatively easy, determining the new loop bounds is more complicated for non-singular transformations than it was for unimodular transformations.

Suppose that $(I_1, I_2, \ldots, I_n)$ is the original iteration space, T is the transformation matrix, and $(J_1, J_2, \ldots, J_n)$ is the transformed iteration space. The substitution in the

assignment statements is determined as follows:

$$(I_1, I_2, \ldots, I_n) = T^{-1}(J_1, J_2, \ldots, J_n)$$

Then, the new values for $I_1, \ldots, I_n$ are substituted into the original assignment statements. Note that this is the case for both unimodular and non-singular matrices.

Transforming the loop bounds for a non-singular matrix is more complex than for a unimodular matrix. The reason for the complication is based upon the value of the determinant of the transformation matrix T ($det\ T \geq 1$). The transformed iteration space may be much more sparse than the original iteration space, and the step sizes of the transformed loops are typically greater than one.

The work presented by Li and Pingali, and Ramanujam relaxes the selection of the transformation matrix T. The relaxation enables T to be more easily selected, and enables the representation of one additional loop transformation technique, *loop scaling*. As discussed previously, the criteria for legality, optimality, and assignment statement mapping are equivalent for unimodular and non-singular matrices. However, the mapping of the loop bounds for the transformed code is more complex because of possible changes in the density of the iteration space. The use of non-singular matrices complicates the loop bound mapping strategy, while adding only limited expressive power (one additional transformation).

8.3 Schedules

Kelly and Pugh introduce a representation of source code transformations called *schedules* that is an alternative to matrix-based representations [7]. Schedules are in the form:

$$T : [i_1, \ldots, i_m] \rightarrow [f_1, \ldots, f_n] | C$$

where T is the schedule; i_l is the original iteration space; f_k is a function of original iteration space; and C represents the domain restrictions. Note that the original iteration space may have a different loop nest depth (m) than the final loop nest (n). In unimodular matrices,

a transformation matrix T is assigned to an entire block of source code. In contrast, each individual assignment statement has its own schedule.

The legality of a schedule is based upon its *variable part* and *constant part.* The variable part is the portion of the schedule that is a function of the original index variables, and the constant part is what remains after the variable part is removed. A *level* refers to the nest depth of the transformed code. As an example, suppose that we are given the schedule:

$$[i, j, k] \rightarrow [i + 2k + n + 1, 3j, 2, 2k + 3n]$$

Then the variable part of the schedule at level 1 is $i + 2k$, and the constant part is $n + 1$.

Legality is formally defined as:

$$\forall i, j, p, q, Sym : i \rightarrow j \in d_{pq} \Rightarrow T_{p(i)} \prec T_{q(j)}.$$

Informally, this definition states that if there is a dependency from between statement p at iteration i and statement q at iteration j in the initial source code, then the transformed code must maintain the same execution order. A legal schedule T that satisfies the execution order constraint is calculated based upon its variable part and constant part. The variable part is algorithmically determined and the constant part is heuristically resolved.

Optimality of a schedule is determined based upon three criteria: *simplicity, granularity,* and *locality* [8]. Simplicity is a metric that is based upon the complexity of the schedule and the amount of parallel execution between statements. Granularity is a measure of the total number of iterations that are executed sequentially outside of the outermost parallel loop. Each loop size is assumed to be 100, since the value may not be known at compile time. Finally, locality is an estimate of cache misses that may occur. Again, each loop is assumed to have 100 iterations, each cache line to contain 8 elements, and array layout is row major.

Schedules are used to map the initial to transformed source code based upon *intervals.* The schedules for each assignment statement at each level of the schedule are compared to

determine whether or not the statement is executing in the interval. Based upon the comparisons, transformed source code is generated and simplified. Schedules are implemented in *Omega/Tiny* [7], which is a data dependence/source code transformation tool that is built on top of *Tiny* [19].

Schedules allow a large family of transformations to be represented. The ability to represent transformations in addition to loop skewing, reversal, and permutation leads to a more usable compiler. Also, the genericity of schedules implies that they are extensible to other loop transformations.

However, large data structures are needed since there exists a unique schedule for each source code statement. Also, the solution set is infinitely large for the variable and constant parts of the schedules. Apparently, some clever heuristic is used in the implementation to prune the search space. As mentioned, an assumption that is made when determining optimality is that the loop size is always 100. In many cases this assumption is incorrect, and extraneous optimality results may be obtained. Finally, and most importantly, the mapping algorithm is exponential ($O(3^n)$) [36], because of the method used to determine intervals.

8.4 Generalized Loop Transformations

Sarkar and Thekkath [9] present work that combines some of the benefits of a formal specification approach with that of matrix representation of loop transformations. They present templates that formally describe the pre- and postconditions for the dependence vector set, the loop bounds, and initialization statements for a kernel set of transformations applied to perfect loop nests. The kernel set of transformations is small, but extensible, and includes *Unimodular, ReversePermute, Parallelize, Block, Coalesce,* and *Interleave.* Each technique is described in terms of a *template* that describes what must be true before the transformation can be applied, the input parameters that are required, and the effect of the transformation on the source code.

The overall transformation of serial to parallel source code occurs in four steps:

1. Template instantiation
2. Sequence representation for loop transformations
3. Legality tests
   - Dependence vector test
   - Loop bounds test
4. Code generation for the transformed loop nest
   - Generate new loop bound expressions
   - Generate new initialization statements

Each of the four steps will be briefly described below.

Template instantiation is obtained by supplying the input parameters, such as the number of nested loops, for one of the transformation techniques. A *Sequence* of transformations is represented by a global sequence $T = \langle t_1, t_2, \ldots, t_k \rangle$, where each t_i represents an individual transformation in the sequence. Note that the concept of a global sequence T has no real meaning in this context. In Sarkar and Thekkath's framework, the effect of T is determined by application of transformations for each of the individual t_i's in the sequence. In contrast, the transformed code can be determined directly from T with no concern as to the individual transformations in the unimodular approach.

The *Legality tests* are divided into two subsections: *Dependence vector test* and *Loop bound test.* The dependence vector test ensures that all dependence vectors are lexicographically positive after transformation. The loop bound test checks the bounds of the loop to determine if the preconditions are met for the individual technique. Each transformation in the kernel set has a unique set of preconditions on the loop bounds.

The final step in the generalized transformation model is *Code generation.* This step describes the effects of the transformations in terms of loop bounds: how the upper and lower loop bound for each loop in the loop nest is affected; and initialization statements: how the index variables are affected.

The transformation template for the dependence vector set for *loop blocking* will be shown for illustrative purposes. The template for the dependence vector set is:

$$D' = \{ \quad (d_1, \ldots, d_{i-1}, d'_i, \ldots, d'_j, d''_i, \ldots, d''_j, d_{j+1}, \ldots, d_n) |$$
$$(d_1, \ldots, d_n) \in D \wedge$$
$$(d'_k, d''_k) \in \mathit{blockmap}(d_k)\ \forall i \leq k \leq j\}$$

where $\mathit{blockmap}(d_k)$ is defined as:

$$\begin{array}{ll} \{(0,0)\} & \text{if } d_k = 0 \\ \{(\text{`}*\text{'}, \text{`}*\text{'})\} & \text{if } d_k = \text{`}*\text{'} \\ \{(0, d_k), (d_k, \text{`}*\text{'})\} & \text{if } d_k = 1 \text{ or } -1 \\ \{(0, d_k), (\mathit{dir}(d_k), \text{`}*\text{'})\} & \text{otherwise} \end{array}$$

and $\mathit{dir}(d_k)$ is defined as:

$$\begin{array}{ll} d_k & \text{if } d_k \text{ is a direction value or if } d_k = 0 \\ \text{“}+\text{”} & \text{if } d_k \text{ is a positive distance value} \\ \text{“}-\text{”} & \text{if } d_k \text{ is a negative distance value} \end{array}$$

Suppose that we are given the following dependence vector set:

$$D = \{(2, 0, +, -1), (3, -2, 1, -1), (+, -1, *, 2)\}.$$

Using the mapping just described for loop blocking and given that the loop nest depth is four ($n = 4$), loops $i = 2$ through $j = 3$ are blocked, and $\mathit{bsize}[1 \ldots 2]$ is a vector representing the block size for loops i through j, then the new dependence vector set is:

$$\begin{aligned} D' = \quad & \{(2,0,0,0,+,-1), (2,0,+,0,*,-1), (3,0,0,-2,1,-1), (3,0,1,-2,*,-1), \\ & (3,-,0,*,1,-1), (3,-,1,*,*,-1), (+,0,*,-1,*,2), (+,-1,*,*,*,2)\} \end{aligned}$$

This approach combines some elements of formal specifications and matrix-based representation of transformations. The transformations are not described in terms of a general framework. The individual transformations (t_i's) must each be applied to determine the global effect of the transformations (T). However, overall legality of the sequence cannot be determined until all of the transformations are applied. Therefore, if one of the early

transformations in a sequence causes an illegal condition, then the compiler must "back up" through the transformation sequence, eliminate the illegal transformation, then reapply the remainder of the sequence. In contrast, in the unimodular approach only the final transformation matrix T is used to translate the initial source code into final form. Thus, if T causes an illegal condition, then it is much easier to correct.

The applicability of a transformation is dependent upon the accuracy of the dependence information. Potential parallelism is determined based upon properties of the transformed dependence vector set. Therefore, if dependence vector set mapping rules result in too restrictive, or spurious dependence vectors, then potential parallelism may be lost. Finally, Sarkar and Thekkath's method requires perfect loop nests.

8.5 Precondition Approach

Whitfield and Soffa [37, 38, 39] propose a formalized method for applying parallel compiler techniques to translate source code. Their work describes a method to determine interdependences between compiler techniques and also discusses methods for exploiting them in order to establish an optimal ordering for compilation. The ordering is based solely upon interactions between compiler methods and is completely independent of the properties of the underlying source code.

First, seven common parallel compilation methods are formally described in terms of pre- and postconditions. The seven code transformations are *Dead Code Elimination, Constant Propagation, Invariant Code Motion, Loop Unrolling, Strip Mining, Loop Fusion,* and *Loop Interchange.* Next, enabling and disabling conditions for each individual method are determined. The enabling and disabling conditions describe what conditions are necessary in order for the compiler method to be applied, and are used to determine any interactions between optimizations. Since each pre- and postcondition is formally described, reasoning about dependences between the individual transformations is also formalized. Finally, enabling and disabling graphs are developed between the seven transformations. The results

from the enabling and disabling graphs are combined into a global directed graph that implies overall transformation ordering.

The next step in this method is to determine the enabling and disabling conditions for each of the seven code transformations. The enabling conditions are the preconditions (i.e., the conditions necessary for the optimization to be applied), and the disabling conditions are negations of the enabling conditions. Then the enabling and disabling conditions are compared in a pairwise fashion to determine enabling and disabling conditions between the transformations. Lack of interactions between code transformations is also proven using formal reasoning techniques.

The final step is constructing an overall dependency graph based upon the pairwise comparison mentioned above. This is a three step process:

1. Develop the enabling graph;
2. Develop the disabling graph;
3. Develop the overall dependency graph (optimization ordering).

Step 3 is a consolidation of steps 1 and 2, and any conflicts arising from step 1 or 2 are reflected in the overall dependency graph.

Finally, the ordering of the techniques is obtained by reviewing the results from the overall dependency graph. Whitfield and Soffa suggest the following order:

1. *Constant Propagation*
2. *Dead Code Elimination* (iterate)
3. *Loop Interchange* (iterate)
4. *Invariant Code Motion* (iterate)
5. *Loop Interchange* and *Invariant Code Motion* (iterate)
6. *Loop Unrolling* (iterate)
7. *Loop Fusion* (iterate)
8. *Strip Mining*

Most of the available parallelism in source code is in the loops. Therefore, extending the transformation set to include more loop transformations may increase the applicability of this approach. There is no metric mentioned to measure parallelism or data locality in the transformed source code with this approach. In most cases, parallelism or data locality is the goal of loop transformations and having a metric shows whether the goals have been met. Another disadvantage of this approach is that static ordering of the transformations is independent of the source code to which the transformations are applied. Different source code and different machines have different optimization goals. Attempting to obtain one ordering of the transformations that meets multiple goals seems optimistic.

8.6 Morphism Notation

Lu and Chen [40] present a mapping from original source code to parallelized code, rather than the transformations themselves. The mappings, or morphisms, are broken down into three subclasses: *spatial, temporal,* and *reordering.*

Temporal morphism is the mapping of the bounding loops of a nested loop. Temporal mapping determines which loops can be parallelized and which loops may be interchanged. It is important to note that the temporal morphism does *not* change the ordering of the statements within the loop, only the order and parallelism of the iterative structures (`do`'s, `for`'s) surrounding the statements.

Reordering maps the statements within the loops into some new ordering. As an example, a statement reordering morphism may indicate that the original loop ordering $(S_1, S_2, \ldots, S_k, \ldots, S_n)$ may be transformed into parallelized code with the following statement order $(S_k, S_2, \ldots, S_n, \ldots, S_1)$.

The final morphism is spatial morphism. Spatial morphisms are hardware dependent translations. In other words, they answer the question "How is the code domain mapped to the available architecture?" An example of a spatial morphism is *strip mining.* Strip mining determines the distribution of data into "chunks" that are dependent upon hardware vector

size. Spatial morphisms were not the focus of Lu and Chen's work and are briefly discussed below.

Assume that D is the domain of the initial source code, m is the transformation mapping, and P is the domain of the transformed (parallelized) source code, then:

$$m(D) \rightarrow P$$

Further, assume that the mapping function m has spatial s, temporal t, and reordering r components, then:

$$m(D) = f(s(D), t(D), r(D)) \rightarrow P$$

That is, the overall mapping is some function of the spatial, temporal, and statement reordering functions over the domain of the initial source code.

The overall transformer that combines the spatial, temporal, and reordering as described above, is called the *loop transformer.* A mapping that combines only temporal and reordering morphisms:

$$\pi(D) = g(t(D), r(D)) \rightarrow P'$$

is called a *loop scheduler.*

The loop transformer m is hardware dependent and the loop scheduler π is hardware independent. Lu and Chen's interest was hardware independent translations and so the focus was loop scheduling rather then overall loop transformations.

Suppose that the initial source code is of the form shown in Figure 8.1(a). A temporal mapping t indicates the order of the bounding loops. Any index name that is *not* included in t is parallelized (`for` $\rightarrow$ `forall`), and must be innermost after transformation. As examples, $t(D) = (k, i, j)$ results in the transformed loop shown in Figure 8.1(b), and $t(D) = (j, k)$ is interpreted as shown in Figure 8.2(a).

The statement reordering function r indicates the order of the statements, by position, after transformation. As an example, $r(D) = (2, 3, 1)$ indicates that the statement that

```
L1  for I1 = ...
L2   for I2 = ...
L3    for I3 = ...
S1
 ⋮
Sn
```

Figure 8.1(a). Sample code: before *morphism* .

```
L1  for I3 = ...
L2   for I1 = ...
L3    for I2 = ...
S1
 ⋮
Sn
```

Figure 8.1(b). Sample code: after *temporal morphism* with $t(D) = (I_3, I_1, I_2)$.

was originally first is now second; originally second is now third; and originally third is now first.

A complete loop schedule for the original example shown in Figure 8.1(a) is:

$$\pi(D) = (t(D), r(D)) = ((j), (1, 3, 2))$$

and results in the loop transformation shown in Figure 8.2(b).

```
L1  for I2 = ...
L2   for I3 = ...
L3    forall I1 = ...
S1     ...
S2     ...
S3     ...
```

Figure 8.2(a). Sample code: after *temporal morphism* with $t(D) = (I_2, I_3)$.

```
L1  for I2 = ...
L2   forall I1 = ...
L3    forall I3 = ...
S1     ...
S3     ...
S2     ...
```

Figure 8.2(b). Sample code: after *temporal* and *reordering morphism* with $\pi(D) = (t(D), r(D)) = ((I_2), (1, 3, 2))$.

An important constraint on transformations is that the resultant dependence vectors are lexicographically positive. This constraint means that any dependences that existed before transformation must still exist after transformation.

A difficulty with this approach is that it only works on perfect loop nests. Any deviation from perfect nesting results in a loop structure that cannot be represented by the notation as described in this paper. Some extension to the mappings that allows imperfect loop nests is needed.

The method of determining how the morphisms are determined was only briefly discussed in the paper and is a non-trivial task. That is, the representation of the mapping was described, but not the derivation of the mapping. Finally, the division of spatial, temporal, and statement reordering morphisms allows the representation of the hardware-independent portion of mapping (loop scheduling) and the hardware-dependent portion of mapping (overall loop transformation).

CHAPTER 9

Conclusions and Future Investigations

This dissertation presented an extension to the kernel set of unimodular transformations, which includes loop permutation, loop skewing, and loop reversal applied to perfect loop nests. Several advantages to the traditional unimodular approach are that it unifies seemingly independent transformations into a common framework; a sequence of transformations is composed into the product of the matrices representing the individual kernel transformations; the source code is mapped from its initial to final form only once, after a final transformation matrix T is found; and parallelism goals are evaluated based upon the product of the transformation matrix T and an abstraction of the source code in the form of a distance vector set D.

The unimodular transformations have been extended to include five other matrix-based transformations [12, 22, 23]. The extended transformations are: loop normalization, which modifies the step size and upper bound of a loop so that they are both one; loop blocking, which includes tiling, strip mining, and cycle shrinking and increases the depth of a nested loop; loop collapsing, which includes loop coalescing and decreases the depth of a nested loop; loop fission, which divides a set of assignment statements within a loop nest into two adjacent loop nests; and loop fusion, which combines two sets of assignment statements from adjacent loop nests into a common loop nest. The legality of the extended set of

transformations was discussed in terms of both perfect and imperfect loop nests. Explicit mapping rules for the distance vector set were described, as well as any modifications to the source code mapping.

Several algorithms were described that show the application of various combinations of transformations from both the kernel and extended transformation set. Finally, the preliminary design for the implementation of a tool that uses the extended transformation set was described. The detailed requirements analysis can be used to guide the development of a parallel compiler that supports the extended, matrix-based model of loop transformations.

Potential areas of future investigations are to develop new algorithms using the extended, matrix-based framework, and remove the constraints on the structure and syntax of the target source code language to which the extended framework is applied.

New algorithms, that include both the extended and kernel set of loop transformations, can be designed to meet other machine-independent and machine-dependent goals. As an example, we did not discuss the application of the extended family of transformations to data distribution for a distributed-memory machine. Now that the unified model has been developed, there are numerous software optimization goals to which it can be applied.

Second, the theoretical model should be extended to include direction vectors, as well as distance vectors. It is often the case that explicit distance information is unavailable because of complexities in the source code dependences. The ability to handle distances, directions, and combinations of the two, could make the theoretic model applicable to a broader range of source code.

We assumed that the loop bounds and subscript expressions were linear functions of the index variables. This assumption is reasonable based upon statistical data derived from actual source code [25]. However, including more complex expressions would broaden the scope of the source code to which the unified model could be applied.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, 1967.

[2] Jau-Hsiung Huang. On Parallel Processing Systems: Amdahl's Law Generalized and some Results on Optimal Design. *IEEE Transactions on Software Engineering*, 18(5):434–448, May 1992.

[3] Uptal Banerjee. Unimodular Transformations of Double Loops. In *Proceeding of the 3rd Workshop on Languages, Compilers, and Parallel Computing*, pages 192–219, August 1989.

[4] Michael Wolf and Monica Lam. A Loop Transformation and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[5] J. Ramanujam. Non-unimodular Transformations of Nested Loops. In *Proceedings of Supercomputing '92*, pages 214–223, November 1992.

[6] Wei Li and Keshav Pingali. A Singular Loop Transformation Framework Based on Non-singular Matrices. In *5th Workshop on Languages and Compilers for Parallel Computing*, pages 249–260, August 1992.

[7] Wayne Kelly and William Pugh. A Framework for Unifying Reordering Transformations. Technical Report CS-TR-2995-1, University of Maryland, August 1993.

[8] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. Technical Report UMIACS-TR-93-67,CS-TR-3108, University of Maryland, August 1993.

[9] Vivek Sarkar and Radhika Thekkath. A General Framework for Iteration-Reordering Loop Transformations. In *Proceeding of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187, June 1992.

[10] David Chesney. A Formal Approach to Automatic Source Code Translation for Parallel Architectures. Technical Report CPS-91-15, Michigan State University, October 1991.

[11] David Chesney and Betty Cheng. Formal Specification of an Automatic Source Code Translator for Parallel Computer Architectures. In *Minnowbrook Workshop on Software Engineering for Parallel Computing*, August 1992.

[12] David R. Chesney and Betty H.C. Cheng. Generalizing the Unimodular Approach. In *International Conference on Parallel and Distributed Systems*, pages 398–403, December 1994.

[13] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[14] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

[15] Hans Zima and Barb Chapman. *Supercompilers for Parallel and Vector Computers.* The ACM Press, 1990.

[16] James Davies, Christopher Huson, Thomas Macke, Bruce Leasure, and Michael Wolfe. The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer. In *International Conference on Parallel Processing*, pages 833–835, 1986.

[17] Ken Kennedy, Kathryn McKinley, and Chau-Wen Tseng. Analysis and Transformation in the Parascope Editor. Technical Report CRPC-TR90106, Rice University, December 1990.

[18] David Padua and Michael Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[19] Michael Wolfe. The Tiny Loop Restructuring Research Tool. In *International Conference on Parallel Processing*, pages II46–II53, 1991.

[20] Zhiyuan Li and Pen-Chung Yew. Interprocedural Analysis for Parallel Computing. In *International Conference on Parallel Processing*, pages 221–228, 1988.

[21] Constantine Polychronopoulos, Milind Girkar, Mohammed Haghighat, Chia Lee, Bruce Leung, and Dale Schouten. *Languages and Compilers for Parallel Computing*, chapter 21 - The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran, pages 423–435. MIT Press, 1990.

[22] David Chesney and Betty Cheng. Application of the Unimodular Approach to Loop Fission and Loop Fusion. Presented at the *Scalable High Performance Computing Conference*, May 1994.

[23] David Chesney and Betty Cheng. Extending the Unimodular Approach to Other Transformation Techniques. Technical Report CPS-93-24, Michigan State University, April 1993.

[24] Michael Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Stanford University, August 1992.

[25] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[26] Ron Sass and Matt Mutka. Transformations on Doubly Nested Loops. In *International Conference on Parallel Architectures and Compilation Techniques*, 1994.

[27] Constantine Polychronopoulos. Loop Coalescing: A Compiler Transformation for Parallel Machines. In *International Conference on Parallel Processing*, pages 235–242, 1987.

[28] Ron Sass and Matt Mutka. Enabling Unimodular Transformations. In *Supercomputing*, pages 753–762, 1994.

[29] Convex Computer Corporation. *Convex FORTRAN Optimization Guide*, second edition, 1994.

[30] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[31] Uptal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic, 1988.

[32] Michael Wolfe and Chau-Wen Tseng. The Power Test for Data Dependence. Technical Report Rice COMP TR90-145, Rice University, December 1990.

[33] Michael Wolfe and Chau-Wen Tseng. The Power Test for Data Dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

[34] W. Pugh. The Omega Test: A Fast and Practical Algorithm for Dependence Analysis. In *Proceedings of Supercomputing '91*, pages 4–13, November 1991.

[35] Michael Wolfe. *TINY: A Loop Restructuring Research Tool.* Oregon Graduate Institute of Science and Technology, December 1990.

[36] Wayne Kelly. A Framework for Unifying Reordering Transformations. personal communication, September 1993.

[37] Debbie Whitfield and Mary Lou Soffa. An Approach to Ordering Optimizing Transformations. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 137–146, March 1990.

[38] Deborah Whitfield and Mary Lou Soffa. Investigating Properties of Code Transformations. In *1993 International Conference on Parallel Processing*, pages II–156–160, August 1992.

[39] Deborah Lynn Whitfield. *A Unifying Approach for Optimizing Transformations.* PhD thesis, University of Pittsburgh, 1991.

[40] Lee-Chung Lu and Marina Chen. New Loop Transformation Techniques for Massive Parallelism. Technical Report YALEU/DCS/TR-833, Yale University, October 1990.