AD-A249 326

DTIC
ELECTE
MAY 1992
S C D

2

A Unified Framework for Systematic Loop Transformations

Lee-Chung Lu and Marina Chen

YALEU/DCS/TR-816
October, 1990

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

92-09999

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

92 4 20 053

Statement A per telecon
Dr. Richard Lau ONR/Code 1111
Arlington, VA 22217-5000

NWW 5/1/92

Accession For
NTIS GRA&I ✓
DTIC TAB ☐
Unannounced ☐
Justification

By
Distribution/
Availability Codes

Dist	Avail and/or Special
A-1	

QUALITY INSPECTED 4

Yale University
Department of Computer Science

A Unified Framework for Systematic Loop Transformations

Lee-Chung Lu and Marina Chen

YALEU/DCS/TR-816
October, 1990

This work has been supported in part by the Office of Naval Research under Contract N00014-89-J-1906, N00014-90-J-1987.

A Unified Framework for Systematic Loop Transformations

Lee-Chung Lu and Marina Chen

Department of Computer Science
Yale University
P.O. Box 2158 Yale Station
New Haven, CT 06520
lu-lee-chung@yale.edu

Abstract

This paper presents a formal mathematical framework which unifies the existing loop transformations. This framework also includes more general classes of loop transformations, which can extract more parallelism from a class of programs than the existing techniques. We classify *schedules* into three classes: *uniform*, *subdomain-variant*, and *statement-variant*. Viewing from the degree of parallelism to be gained by loop transformation, the schedules can also be classified as *single-sequential level*, *multiple-sequential level*, and *mixed* schedules. We also illustrate the usefulness of the more general loop transformation with an example program.

1 Introduction

One of the central issues in restructuring compiler is to discover parallelism automatically and generate correct parallel control structures that can take advantage of the large number of processors. The advent of massively parallel machines opens up opportunities for programs that have large-scale parallelism to gain tremendous performance over those that do not.

This paper presents a formal mathematical framework which unifies the existing loop transformations such as loop interchanging [1, 2, 17, 19] permutation [3], skewing [17, 19], reversal, the wavefront method [7, 9, 10, 11, 13, 14, 15], and statement reordering. This framework also includes more general classes of loop transformations which can extract more parallelism from a class of programs than the existing techniques. The particular class of programs are those that consist of perfectly nested loops possibly with conditional statements where the guards as well as the array index expression are affine expressions of the loop indices.

In the next section, we describe the notations and terminologies used in the paper. We then present a formal mathematical framework which unifies the existing loop transformation techniques, and sets the stage for discussing the more general classes of *loop transformers* in Section 3. A *loop transformer* is a function that relates a given loop nest with its transformed version, and consists of two parts: a *spatial morphism*, and a *temporal morphism*, called a *schedule*. Next, in Section 4, we classify schedules, by the properties of *uniformity*, into three classes: *uniform*, *subdomain-variant*, and *statement-variant*. Viewing from the degree of parallelism to be gained by loop transformation, the schedules can also be classified as *single-sequential level*, *multiple-sequential level*, and *mixed* schedules. We also describe the functional forms of the schedules for each class. Existing loop transformation techniques are given as examples of these classes of schedules.

Due to the limited space, please refer to [12] for the algorithms for obtaining the more general classes of schedules. The problem formulations for obtaining these schedules are based on

[0]in Proceedings of the Third ACM SIGPLAN Symposium on PPOPP, April 1991.

dependence index pairs, which provide more dependence information than *dependence vectors*. Since there are many such pairs that need to be considered, and they can be infinitely many when the loop bounds are unknown at compile time, we need to rely on a technique called *polyhedra decomposition* [8, 15] to manage the complexity of the algorithm. In addition, nonlinear programming and bounded enumerative search are required to obtain optimal schedules. The complexity of nonlinear programming is reduced by using fast heuristics and linear programming as described in [12], which obtain optimal schedules for most cases.

Finally, we illustrate the usefulness of the more general loop transformations with an example program in Section 5. Versions of the transformed program using different schedules are implemented on a Connection Machine CM/2. The difference in performance, which is essentially due to the available parallelism determined by the schedule, can amount to two orders of magnitude.

2 Definitions and Terminologies

Throughout this paper, programming examples are written in a Fortran-like notation although the transformation techniques also apply to functional languages.

Index Domains Let $[a, b]$ be an *interval domain* of integers from a to b. We define an *index domain* D (also called an *iteration space* in [17]) of a d-level *perfectly nested* loop

Loop Nest 1

```
DO (i_1 = l_1, u_1) {
  DO (...) {
    DO (i_d = l_d, u_d) {
      body }          } }
```

to be the Cartesian product $[l_1, u_1] \times \ldots \times [l_d, u_d]$ of d interval domains $[l_k, u_k]$ for $1 \leq k \leq d$.

For the purpose of formulating loop transformations, we consider D to be a subset of the d-dimensional vector space over rationals. Throughout the paper, we let $I = (i_1, \ldots, i_d)$ and $J = (j_1, \ldots, j_d)$. With the domain and tuple notations, Loop Nest 1 can be rewritten as follows:

```
DO (I:D) {
  body }
```

In this paper, we focus on sequential loop nests which are perfectly nested. We use the following loop nest as a generic example throughout the paper, where D is a d dimensional index domain and $\tau[a]$ is an expression containing a:

Loop Nest L (Generic Loop Nest)

```
DO (I:D) {
  ...
  S_1 : IF(P_1) A(X(I)) = ...
  ...
  S_2 : IF(P_2) B(Z(I)) = τ[A(Y(I))]
  ...                                }
```

Data Dependence We now define dependence between statements. Let S_1 and S_2 be two statements of a program. A *flow dependence* exists from S_1 to S_2 if S_1 writes data that can subsequently be read by S_2. An *anti-dependence* exists from S_1 to S_2 if S_1 reads data that S_2 can subsequently overwrite. An *output dependence* exists from S_1 to S_2 if S_1 writes data that S_2 can subsequently overwrite. We use the notation $S_1 \Rightarrow S_2$ to denote a dependence from S_1 to S_2.

Consider Loop Nest L. For statement S_2 to compute the value $B(Z(J))$ at iteration J, the value $A(Y(J))$ is needed. If $A(Y(J))$ is computed from statement S_1 at iteration I, i.e. $Y(J) = X(I)$, then we say S_2 at iteration J is flow dependent on S_1 at iteration I, denoted by $S_1@I \Rightarrow S_2@J$.

3 Formalizing Loop Transformation

We now formalize the notion of loop transformation from a source loop nest to a target parallel loop nest. A *loop transformer* is a function defined over the Cartesian product of the iteration space of the loop nest and the set of statements in the body of the loop that relates a given loop nest with its transformed version. From the standpoint of symbolic transformation of the program text, a loop transformer can be decomposed into two components: the first component, called *domain morphism*, defines how the iteration space should be mapped to a new one (with new loop bounds and possibly new predicates guarding the loop body), and the second component, called *statement reordering function*, defines the ordering of the statements in the transformed loop nest. The process of obtaining a loop transformer, however, suggests another decomposition: a *temporal morphism* and a *spatial morphism*.

3.1 Loop Transformer and Schedule

Kinds of Index Domains For the purpose of loop transformation, it is useful to indicate how the index domain shall be interpreted. We do this by defining *kinds* of index domains. The kind of an interval domain D can be either *spatial* or *temporal*. The kind of a product domain is the product of the kinds of the component domains. For example, $D_1 \times D_2$ is of kind temporal×spatial if D_1 is of kind temporal and D_2 is of kind spatial. A single-level loop with a temporal index domain corresponds to a sequential loop (i.e. DO), while a spatial index domain corresponds to a parallel loop (i.e. DOALL).

Lexicographical Ordering We use the following notations to denote lexicographical ordering on elements X and Y of an n-dimensional index domain. We define "$\prec$" to be the lexicographical ordering: we say $X \prec Y$ if there exists k, $1 \leq k \leq n$, such that $x_l = y_l$ for all l, $l < k$, and $x_k < y_k$. Similarly, we say $X \preceq Y$ if $X \prec Y$ or $x_k = y_k$ for all k, $1 \leq k \leq n$. We use $\hat{0}$ to denote the zero vector.

Domain Morphism We define a *domain morphism* to be a bijective function g from index domain D to index domain E, denoted by $g{:}D \rightarrow E$, such that for all dependences $S_1@I \Rightarrow S_2@J$, condition $g(J) - g(I) \succeq \hat{0}$ holds. In other words, a domain morphism will never reverse the ordering imposed by dependence relations.

In this paper, we restrict the codomain E of a domain morphism to be a cross product of a temporal index domain E_1 and a spatial index domain E_2, i.e. $E = E_1 \times E_2$. Under this restriction, all parallel loops are innermost loops in the transformed loop nest. We define g_1 and g_2 to be two functions:

$$\begin{aligned} g_1 &: D \rightarrow E_1 \\ &\quad \text{(called a \textit{temporal morphism}), and} \\ g_2 &: D \rightarrow E_2 \\ &\quad \text{(called a \textit{spatial morphism}).} \end{aligned} \tag{1}$$

Under domain morphism g, index I in the original loop will be mapped to index $J = g(I)$ in the transformed loop nest. Since g is bijective, it has a well-defined inverse, denoted by g^{-1}. Clearly, $I = g^{-1}(J)$. The following loop nest

Loop Nest 2

```
DO ((I:D)){
   ... A(X(I)) ...}
```

will be transformed into the following new loop nest under domain morphism $g{:}D \rightarrow E_1 \times E_2$:

Loop Nest 3

```
DO ((J1:E1)){
  DOALL ((J2:E2)){
     ... A(X(g^{-1}(J1 : J2))) ...} }
```

where $(J_1 : J_2)$ denotes the concatenation of two vectors J_1 and J_2.

The requirement of g to be surjective is in fact not essential. For any injective function $g'{:}D \rightarrow E$, we can always derive a corresponding bijective function $g{:}D \rightarrow \{g'(I) \mid I \in D\}$ from D to the image of D under g' [6]. Therefore, by allowing the codomain of a bijective function to be the image of an injective function, we allow a much more general class of functions to be used as domain morphism. For comparison, the unimodular transformations discussed in [4, 16] are special classes of bijective functions. The generality does require some nontrivial algebraic manipulation to generate correct loop bounds and predicates to guard the conditional statements in the transformed loop nest. An automatic transformation procedure for doing this based on an equational theory is described in [6].

Statement Reordering We now discuss statement reordering. Let $\mathcal{S}$ denote the set of statements in the loop body. We define a *statement reordering* to be a function h from the set of statements to the set of statement labels:

$$r{:}\mathcal{S} \rightarrow [0, s-1], \tag{2}$$

where $s = |\mathcal{S}|$, the number of statements in $\mathcal{S}$.

Loop Transformer With g and r defined above, the following function h, called the *loop transformer*, specifies how a loop nest is transformed:

$$\begin{aligned} &h{:}D \times \mathcal{S} \rightarrow E_1 \times E_2 \times [0, s-1] \\ &h(I, S) = (g_1(I), g_2(I), r(S)). \end{aligned} \tag{3}$$

Schedule Given h defined above, a *schedule* π is defined to be a function

$$\begin{aligned} &\pi{:}D \times \mathcal{S} \rightarrow E_1 \times [0, s-1] \\ &\pi(I, S) = (g_1(I), r(S)), \end{aligned} \tag{4}$$

such that condition $\pi(J, S_2) - \pi(I, S_1) \succ \hat{0}$ must hold for all dependences $S_1@I \Rightarrow S_2@J$ in the loop nest. The condition ensures that the ordering imposed by dependence relations is preserved. Clearly, a schedule determines the sequential execution of the transformed parallel loop nest. Note that by the definition of domain morphism, $g_1(J) - g_1(I)$ can be equal to the zero vector, i.e. $S_1@I$ and $S_2@J$ can be computed at the same iteration in the transformed loop nest. In this case, statement S_1 must be in front of statement S_2 in the loop body, i.e. condition $r(S_1) < r(S_2)$ must hold, to preserve the dependence ordering.

3.2 Overall Procedure to Obtain a New Loop Nest

Finding a schedule π is to understand what is the potential parallelism that can be extracted from the source program. The algorithms for obtaining a schedule π is presented in [12]. The so-called *strip mining* [17] and *tiling* [16, 18] of loops are captured by the spatial morphism g_2. Given a schedule $\pi = (g_1, r)$, the choice of g_2, which depends on factors such as memory and processor organization and communication cost, should keep a loop transformer $h = (g_1, g_2, r)$ injective. A default g_2, which is used in the rest of this paper, can be $g_2(i_1, \ldots, i_d) = (i_{p_1}, \ldots, i_{p_n})$, so as to result in a loop transformer h that is injective, where n is the dimensionality of the spatial index domain E_2, $\{p_1, \ldots, p_n\}$ is a subset of interval domain $[1, d]$, and $p_1 < \ldots < p_n$.

Overall Procedure To summarize, the overall procedure to obtain a new loop nest is:

1. First generate a schedule $\pi = (g_1, r)$ to maximize the degree of parallelism by using the algorithms presented in [12].
2. Then determine the spatial morphism g_2 of domain morphism based on target machine characteristics such as memory and processor organization, communication cost, etc., or use a default function as shown above.
3. The loop transformer is simply $h = (g_1, g_2, r)$.
4. Finally perform symbolic program transformation, given the source loop nest and loop transformer h, to obtain the new loop nest. For the formal procedure, please refer to [6].

We now discuss different classes of schedules which include the exiting schedules in one class.

4 Classes of Affine and Piece-Wise Affine Schedules

We call a schedule affine if it is an affine function of the loop indices. We call a schedule piecewise affine if the restriction of the function to each subdomain of D and each subset of $\mathcal{S}$ is affine. In the loop restructuring literature, only affine schedules are considered. In this paper, we consider, in addition, piece-wise affine schedules.

We now classify schedules according to two properties: (1) the uniformity of the schedule with respect to the the set of statements $\mathcal{S}$ and the index domain D, and (2) the degree of parallelism in the transformed Loop Nest.

4.1 Properties of Schedules

Uniformity Let index domain D be partitioned into m disjoint subdomains D_k, $1 \leq k \leq m$; and let the set of statements $\mathcal{S}$ be partitioned into n disjoint subsets $\mathcal{S}_k$, $1 \leq k \leq n$. The general form of a piece-wise affine schedule π defined in Equation (4) consists of conditional branches, one for each pair of subdomain D_i and statement subset $\mathcal{S}_j$, and an affine expression of the loop indices is on the right-hand side of each branch. We call a schedule

1. *uniform* if $m = 1$ and $n = 1$,
2. *subdomain-variant* if $m > 1$ and $n = 1$, (also called a subdomain schedule)
3. *statement-variant* if $m = 1$ and $n > 1$, or
4. *nonuniform* if $m > 1$ and $n > 1$.

Degree of Generated Parallelism As defined in Equations (1) and (4), the dimensionality of E_1, the temporal index domain, indicates the number of levels of sequential loops in the transformed loop nest. Hence a schedule π would generate a target loop nest with more levels of parallel loops and thus potentially more parallelism if E_1 is of lower dimensionality. We call the dimensionality of E_1 the *sequential level* of π. Schedules can thus be classified as:

1. *Single-sequential level schedule* (SSL) if E_1 is a subset of the set of natural numbers $\mathcal{N}$.
2. *Multiple-sequential level schedule* (MSL) if E_1 is a subset of $\mathcal{N}^n$, where n is a positive integer and $n \leq d$, the dimensionality of the original loop nest.
3. *Mixed schedule* (Mixed) if E_1 can be of different dimensions for each pair of subdomain D_i and statement subset $\mathcal{S}_j$. Such a mixed schedule will result in transformed programs consisting of imperfectly nested loops.

4.2 Classification and Functional Form of Schedules

Classification Clearly, the uniformity of π and the dimensionality of π are two orthogonal properties, except that a mixed schedule cannot be uniform. Thus there are all together eleven $(4 * 3 - 1)$ classes of affine and piece-wise affine schedules. The classes and their acronyms ranging from single-sequential level uniform schedules to mixed nonuniform schedules are in Figure 1.

Functional Form We now describe the forms of affine and piece-wise affine schedules by using matrix and vector notations. Let $r(S)$ for a given S in $\mathcal{S}$ be a constant scalar. Let d be the dimensionality of the index domain of the source loop nest.

Uniform Schedule:

$$\pi(I,S) = (TI, r(S)), \quad I \in D, S \in \mathcal{S}, \tag{5}$$

where T is a constant l-by-d matrix and l is the sequential level of the schedule π.

Subdomain Schedule:

$$\pi(I,S) = \left\{ \begin{array}{l} I \in D_1 \rightarrow (T_1 I, r_1(S)) \\ \dots \\ I \in D_m \rightarrow (T_m I, r_m(S)) \end{array} \right\}, \quad I \in D, S \in \mathcal{S}, \tag{6}$$

where T_i, $1 \leq i \leq m$, is a constant l_i-by-d matrix and l_i is the sequential level of the part of the schedule defined over D_i.

	Single-Sequential Level (SSL)	Multiple-Sequential Level (MSL)	Mixed
Uniform (U)	SSL-U	MSL-U	
Subdomain (SD)	SSL-SD	MSL-SD	Mixed-SD
Statement-Variant (SV)	SSL-SV	MSL-SV	Mixed-SV
Nonuniform (NU)	SSL-NU	MSL-NU	Mixed-NU

Figure 1: Classes of schedules

Statement-Variant Schedule:

$$\pi(I,S) = \left\{ \begin{array}{l} S \in \mathcal{S}_1 \rightarrow (T_1 I, r(S)) \\ \dots \\ S \in \mathcal{S}_n \rightarrow (T_n I, r(S)) \end{array} \right\}, \quad (7)$$
$$I \in D, S \in \mathcal{S},$$

where T_i, $1 \leq i \leq n$, is a constant l_i-by-d matrix and l_i is the sequential level of the part of the schedule defined over $\mathcal{S}_i$.

Nonuniform Schedule:

$$\pi(I,S) = $$
$$\left\{ \begin{array}{l} I \in D_1, S \in \mathcal{S}_1 \rightarrow (T_{11} I, r_1(S)) \\ \dots \\ I \in D_m, S \in \mathcal{S}_n \rightarrow (T_{mn} I, r_m(S)) \end{array} \right\}, \quad (8)$$
$$I \in D, S \in \mathcal{S},$$

where T_{ij}, $1 \leq i \leq m$ and $1 \leq j \leq n$, is a constant l_{ij}-by-d matrix and l_{ij} is the sequential level of the part of the schedule defined over D_i and $\mathcal{S}_j$.

The linear term TI, $I \in D$, determines the form of the sequential loops in the transformed loop nest, which includes nesting structures, bounds, and possibly additional predicates to guard the loop body. The constant terms $r(S)$ determine the orders of the statements in the transformed loop body.

4.3 Examples of Different Classes of Schedules

We now give some examples of different classes of schedules. We first show that loop interchanging, permutation and skewing are special cases of MSL uniform schedules.

Example 1: Loop Interchanging and Permutation Loop interchanging and loop permutation [1, 2, 3, 17, 19] is a process of switching inner and outer loops. Suppose Loop Nest 1 after loop interchanging or loop permutation becomes

Loop Nest 4

```
DO ($i_{p_1} = l_{p_1}, u_{p_1}$){
  DO (...){
    DO ($i_{p_d} = l_{p_d}, u_{p_d}$){
      body }          } },
```

where $(p_1, p_2, \dots, p_d)$ is a permutation of $(1, 2, \dots, d)$. Also suppose the m innermost loops are parallelizable. The schedule π has the form:

$$\pi(I,S) = (i_{p_1}, i_{p_2}, \dots, i_{p_{d-m}}, \mathsf{loc}(S)), \quad (9)$$
$$\text{i.e.} \quad T = \begin{pmatrix} V(p_1) \\ \dots \\ V(p_{d-m}) \end{pmatrix}, \text{ and} \quad (10)$$
$$r(S) = \mathsf{loc}(S), \quad (11)$$

where loc is a function from $\mathcal{S}$ to $\mathcal{N}$ that returns the position of the statement S in the source loop nest, and each $V(k)$ is a vector of length d with k-th element being 1 and all other elements being 0.

Example 2: Loop Skewing This operation transforms Loop Nest 1 as follows: shifting index i_n with respect to index i_m, $1 \leq m < n \leq d$, by a factor of f, where f is a positive integer, replacing l_n with the expression $(l_n + i_m * f)$, replacing u_n with the expression $(u_n + i_m * f)$, and replacing all occurrences of i_n in the loop with the expression $(i_n - i_m * f)$ [17, 19]. The transformed loop nest is of the form:

Loop Nest 5

```
DO ($i_1 = l_1, u_1$){
  ...
  DO ($i_n = l_n + i_m * f, u_n + i_m * f$){
    ...
    DO ($i_d = l_d, u_d$){
      loop body with $i_n$being
      replaced by ($i_n - i_m * f$) } }     }
```

The schedule for loop skewing is of the form:

$$\pi(I,S) = (i_1, \ldots, i_m, \ldots, \underbrace{i_n + f * i_m}_{n\text{-th element}}, \ldots, i_d, \mathrm{loc}(S)), \quad (12)$$

$$\text{i.e. } T = \begin{pmatrix} V(1) \\ \ldots \\ V(n) + f * V(m) \\ \ldots \\ V(d) \end{pmatrix}, \text{ and} \quad (13)$$

$$r(S) = \mathrm{loc}(S), \quad (14)$$

where $\mathrm{loc}(S)$ and $V(k)$ are the same as defined in Example 1.

Example 3: SSL Uniform Schedule

Loop Nest 6

```
DO ($i = 1, n$){
  DO ($j = 1, n$){
    $S_1 : A(i,j) =$
      $B(i, j-1) + i$
    $S_2 : B(i,j) =$
      $A(i-1, j) + j$ } }
```

An SSL uniform schedule

$$\begin{aligned} \pi((i,j,k),S_1) &= (i,1), \quad \text{and} \\ \pi((i,j,k),S_2) &= (i,0), \end{aligned} \quad (15)$$

will transform Loop Nest 6 into

Loop Nest 7

```
DO ($i = 1, n$){
  DOALL ($j = 1, n$){
    $S_2 : B(i,j) =$
      $A(i-1, j) + j$
    $S_1 : A(i,j) =$
      $B(i, j-1) + i$ } }
```

Example 4: MSL Uniform Schedule

Loop Nest 8

```
DO ($i = n-1, 1, -1$){
  DO ($j = i+1, n$){
    DO ($k = i, j$){
      $S_1$ : IF($i + 1 = k$)
        $B(i,j,k) = C(i+1, j, j)$
      $S_2$ : IF($i + 1 < k$)
        $B(i,j,k) = B(i+1, j, k)$
      $S_3$ : IF($i + j + 1 < 2k$)
      $C(i,j,k) = C(i,j,k-1) + B(i,j,k)$ } } }
```

A 2-SL uniform schedule

$$\pi((i,j,k),S) = ((-i,k), \mathrm{loc}(S)) \quad (16)$$

will transform Loop Nest 8 into Loop Nest 9.

Example 5: Mixed Statement-Variant Schedule Consider Loop Nest 8 again. The following schedule transforms Loop Nest 8 to Loop Nest 10, which consists of imperfectly nested loops:

$$\pi((i,j,k),S) = \begin{Bmatrix} S = S_3 \rightarrow \\ \quad ((-i,k), \mathrm{loc}(S)) \\ \textbf{else} \rightarrow \\ \quad (-i, \mathrm{loc}(S)) \end{Bmatrix} \quad (17)$$

Example 6: SSL Subdomain Schedule Another possible transformation of Loop Nest 8

Loop Nest 9

```
DO ($i = 1 - n, -1$) {
  DO ($k = -i, n$) {
    DOALL ($j = 1 - i, n$) {
      $S_1$ : IF($((-i + 1 = k) \wedge (k \leq j)) B(-i, j, k) = C(1 - i, j, j)$
      $S_2$ : IF($((-i + 1 < k) \wedge (k \leq j)) B(-i, j, k) = \tilde{B}(1 - i, j, k)$
      $S_3$ : IF($((-i + j + 1 < 2k) \wedge (k \leq j)) C(-i, j, k) = C(-i, j, k - 1) + B(-i, j, k)$ } } }
```

Loop Nest 10

```
DO ($i = 1 - n, -1$) {
  DOALL (($j = 1 - i, n$), ($k = -i, n$)) {
    $S_1$ : IF($((-i + 1 = k) \wedge (k \leq j)) B(-i, j, k) = C(1 - i, j, j)$
    $S_2$ : IF($((-i + 1 < k) \wedge (k \leq j)) B(-i, j, k) = B(1 - i, j, k)$ }
  DO ($k = -i, n$) {
    DOALL ($j = 1 - i, n$) {
      $S_3$ : IF($((-i + j + 1 < 2k) \wedge (k \leq j)) C(-i, j, k) = C(-i, j, k - 1) + B(-i, j, k)$ } } }
```

Loop Nest 11

```
DO ($t = 2, 2n - 2$) {
  DOALL ($i = n - 1, 1, -1$) {
    DOALL ($j = i + 1, n$) {
      $S_{11}$ : IF($((2t + 3i - 3j > 0) \wedge (t + i - j - 1 = 0))$
        $B(i, j, t + 2i - j) = C(i + 1, j, j)$
      $S_{12}$ : IF($((2t + 3i - 3j \geq 0) \wedge (t + 2i - 2j + 1 = 0))$
        $B(i, j, -t - i + 2j) = C(i + 1, j, j)$
      $S_{21}$ : IF($((2t + 3i - 3j > 0) \wedge (t + i - j - 1 > 0))$
        $B(i, j, t + 2i - j) = B(i + 1, j, t + 2i - j)$
      $S_{22}$ : IF($((2t + 3i - 3j \geq 0) \wedge (t + 2i - 2j + 1 < 0))$
        $B(i, j, -t - i + 2j) = B(i + 1, j, -t - 2i + 2j)$
      $S_{31}$ : IF($((2t + 3i - 3j > 0) \wedge (2t + 3i - 3j - 1 > 0))$
        $C(i, j, t + 2i - j) = C(i, j, t + 2i - j - 1) + B(i, j, t + 2i - j)$
      $S_{32}$ : IF($((2t + 3i - 3j \geq 0) \wedge (2t + 3i - 3j + 1 < 0))$
        $C(i, j, -t - i + 2j) = C(i, j, -t - i + 2j - 1) + B(i, j, -t, -i + 2j)$ } } }
```

is the schedule:

$$\pi((i,j,k),S) = \left\{ \begin{array}{l} i+j-2k < 0 \rightarrow \\ \quad (-2i+j+k, \mathsf{loc}(S)) \\ i+j-2k \geq 0 \rightarrow \\ \quad (-i+2j-k, \mathsf{loc}(S)) \end{array} \right\}, \quad (18)$$

which transforms Loop Nest 8 into Loop Nest 11. Since there are two affine functions for disjoint subdomains of the index domain of the loop nest, each statement in Loop Nest 8 results in two guarded statements in the transformed loop nest. In fact Loop Nest 8 is part of the dynamic programming code presented in Section 5. As one can see, an SSL subdomain schedule can result in code of considerable complexity. It would be a very tedious and error-prone process for a user to write the code by hand. But a compiler can generate the new loop nest, given the schedule, and the original loop nest mechanically.

5 An Application: Dynamic Programming

To illustrate the usefulness of the more general schedules, we take dynamic programming as an example, which has sequential complexity $O(n^3)$ for a problem of size n. The source code is given in Loop Nest 12.

Loop Nest 12

```
DO ($i = 1, n-2$) {
  DO ($j = i+2, n$) {
    $C(i,j) = \min_{i<k<j}$
      $(h(C(i,k), C(k,j)))$ } }
```

This source program is first transformed in a systematic manner by applying *fan-in* and *fan-out* reductions [5] to reduce potential concurrent accesses of variables. The result is Loop Nest 13. Then the code is transformed into three *lisp programs on the Connection Machine CM/2, each with the control structure generated by a 2-SL uniform schedule, a mixed statement-variant schedule and an SSL subdomain schedule respectively. We also have a sequential Common-Lisp program on the Symbolics to compute the same problem. The three schedules are given below. For simplicity, we do not give the constant terms $r(S)$ of function π.

2-SL uniform schedule:

$$\pi(S,(i,j,k)) = (j-i, k-i) \quad (19)$$

mixed statement-variant schedule:

$$\pi(S,(i,j,k)) = \left\{ \begin{array}{l} S = S_{c2} \rightarrow \\ \quad (j-i, k-i) \\ \mathsf{else} \rightarrow j-i \end{array} \right\} \quad (20)$$

SSL subdomain schedule:

$$\pi(S,(i,j,k)) = \left\{ \begin{array}{l} i+j-2k < 0 \rightarrow \\ \quad -2i+j+k \\ i+j-2k \geq 0 \rightarrow \\ \quad -i+2j-k \end{array} \right\}. \quad (21)$$

Experimental Result The experiment is conducted as follows: we run the sequential code on the Symbolics and parallel codes on an 8K-processor Connection Machine with Symbolics as its host. The results described in Figure 2 and Figure 3 show that the version using an SSL subdomain schedule is three orders of magnitude faster than the sequential code, and is two orders of magnitude faster than the versions using a 2-SL uniform schedule and mixed statement-variant schedule. And the program using a mixed statement-variant schedule is about three to four times faster than the program using a 2-SL uniform schedule.

6 Concluding Remarks

We present in this paper a formal mathematical framework which unifies the existing loop transformations. We also present more general affine and piece-wise affine schedules which can extract more parallelism from a class of programs than the existing techniques. The particular class of programs are those that consist of perfectly nested loops possibly with conditional statements where the guards as well as the array index expression are affine expressions of the loop indices. Although the complexity for ob-

Loop Nest 13

```
DO ($i = n-1, 1, -1$) {
  DO ($j = i+1, n$) {
    $m = (i+j+1)/2$
    DO ($k = i, j$) {
      $S_{a1}$ : IF($k < j$) $A(i,j,k) = A(i,j-1,k)$
      $S_{b1}$ : IF($i+1 = k$) $B(i,j,k) = C(i+1,j,j)$
      $S_{b2}$ : IF($i+1 < k$) $B(i,j,k) = B(i+1,j,k)$
      $S_{c1}$ : IF($m = k$) $C(i,j,k) = h_1(A(i,j,k), B(i,j,k),$
        $A(i,j,i+j-k), B(i,j,i+j-k))$
      $S_{c2}$ : IF($m < k < j$) $C(i,j,k) = h_2(C(i,j,k-1), A(i,j,k),$
        $B(i,j,k), A(i,j,i+j-k), B(i.j,i+j-k))$
      $S_{c3}$ : IF($k = j$) $C(i,j,k) = C(i,j,k-1)$
      $S_{a2}$ : IF($k = j$) $A(i,j,k) = C(i,j,k)$                    } } }
```

n	3-SL sequential	2-SL uniform	mixed statement-variant	SSL subdomain
32	6.8	10.72	2.47	0.87
64	55.0	42.88	9.73	1.73
128	440.0	171.50	39.16	3.48
256	3520.0	686.45	235.70	6.96
512	28160.0	2745.80	1159.24	31.70

Figure 2: Running time in seconds.

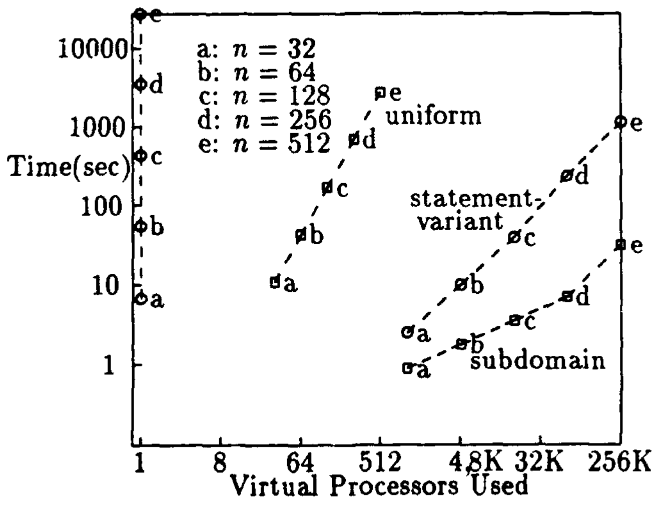

Figure 3: Running time vs. problem size.

taining these more general schedules is high [12], we show that the generated code derived from a new schedule can be two orders of magnitude faster than the version from the existing transformations. For programs not in this particular class, e.g. programs with pointers, *compiler directives* can be added into the sequential programs to help the compiler to generate efficient parallel codes.

References

[1] J.R. Allen. *Dependence Analysis for Subscript Variables and Its Application to Program Transformation.* PhD thesis, Rice University, April 1983.

[2] J.R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] U. Banerjee. A theory of loop permutation. Technical report, Intel Corporation, 1989.

[4] U. Banerjee. Unimodular transformations of double loops. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.* UC. Irvine, 1990.

[5] M.C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, December 1986.

[6] M.C. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 1(2):171–207, July 1988.

[7] J.M. Delosme and I.C.F. Ipsen. Systolic array synthesis: Computability and time cones. Technical Report RR-474, Yale University, 1986.

[8] F. Fernandez and P. Quinton. Extension of Chernikova's algorithm for solving general mixed linear programming problems. Technical Report 437, INRIA-Rennes, October 1988.

[9] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[10] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, and D.V.B. Rao. Wavefront array processor: Language, architecture, and applications. *IEEE Trans. on Computers*, 31(11):1054–1066, November 1982.

[11] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[12] L.C. Lu and M.C. Chen. New loop transformation techniques for massive parallelism. Technical Report TR-833, Yale University, October 1990.

[13] W.L. Miranker and A. Winkler. Spacetime representations of computational structures. In *Computing*, volume 32, pages 93–114, 1984.

[14] D.I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, 71(1), 1983.

[15] P. Quinton and V.V. Dongen. The mapping of linear recurrence equations on regular arrays. Technical Report 485, INRIA-Rennes, July 1989.

[16] M.E. Wolf and M.S. Lam. Maximizing parallelism via loop transformations. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.* UC. Irvine, 1990.

[17] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October 1982.

[18] M. Wolfe. More iteration space tiling. In *Proc. Supercomputing '89*, November 1989.

[19] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.