Program Transformation with Piecewise Linear Domain Morphism

Young-il Choo and J. Allan Yang

YALEU/DCS/RR-985
September 1993

Program Transformation with Piecewise Linear Domain Morphism

Young-il Choo J. Allan Yang

Department of Computer Science
Yale University
New Haven, CT 06520-2158
choo@cs.yale.edu yang@cs.yale.edu

July 24 1993

1 Introduction

The effective use of large-scale parallel computers often requires sophisticated optimizations that change data distribution in order to minimize inter-process communication. For regular parallel algorithms, the pattern of data dependencies can be used in transforming programs so as to minimize non-local data references. Many techniques have been developed to handle situations where the reference patterns, expressed as functions over the index or loop variables, are uniform, linear or affine. A good survey and many references may be found in [4].

In this paper we present a framework of program transformation based on an equational theory of programs in which the data dependence structure of a computation can be transformed in a piecewise linear manner. We begin by describing the Crystal language and how a purely functional notation can be used to describe regular algorithms, then we present the underlying approach to algebraic program transformation, and then conclude with a detailed example of an application where we apply a piecewise linear transformation to Warshall's transitive closure program and derive a more efficient program for a mesh of processors.

2 The Crystal Language

Crystal is a simple functional language based on the lambda notation [2, 1] with basic data types Booleans, integers and floating point numbers. In contrast to other functional languages, Crystal has a special type constructor for defining intervals of integers and subsets of cartesian product of the integers.

2.1 Index Domains

An *index domain* is a type consisting of a set of index points with operations over them. The most basic is the interval index domain elements $\mathsf{interval}(m, n)$, which consists of $n - m + 1$ integers from m to n, with the usual arithmetic operations.

More complicated index domains can be constructed by cartesian product, coproduct, and restriction. The *cartesian product* of index domains D_1 and D_2, denoted $D_1 \times D_2$, is the index domain with whose elements are the ordered pairs of elements from D_1 and D_2, respectively.

The *coproduct* (or *disjoint union*) of D_1 and D_2, denoted $D_1 + D_2$, is the index domain whose elements are $\iota_1(x)$ and $\iota_2(y)$ with x from D_1 and y from D_2 where $\iota_i : D_i \rightarrow (D_1 + D_2)$ are the *injections* into the first and second summand, respectively. When D_1 and D_2 are disjoint, then we dispense with an explicit injection and identify, for example, x with $\iota_1(x)$.

The *restriction* of an index domain D by a *filter* ψ (a Boolean function over D), denoted $D|\psi$, is a new index domain consisting of elements of D which satisfy ψ. For example, $\mathsf{interval}(1, 20)|\mathsf{even?}$ denotes an index domain containing only the even integers between 1 and 20.

Mappings between index domains play a special role in the transformation theory of Crystal and are called *index domain morphisms* or just *domain morphisms.* The identity domain morphism on D is written id_D.

2.2 Expressions

Expressions denote either values, built up using the standard arithmetic and Boolean operators, or types, built up with the type constructors from the basic types and from index domains. The conditional expression has the form

$$\left\{ \begin{array}{c} b_1 \rightarrow e_1 \\ \vdots \\ b_n \rightarrow e_n \end{array} \right\}$$

where the b_i's are Boolean expressions and e_i's are any expressions. For convenience, we allow the keyword **else** to appear as one of the b's, in which case it is interpreted to be the conjunction of the negation of the other b_i's.

Functions are written $\lambda(x_1, \ldots, x_n){:}D.e$, where x_i are identifiers, e is an expression, and an optional type expression T. The dot (.) indicates that the body of the λ-expression should extend as far to the right as possible (and still be well formed). Function application is written in the usual way $f(e)$. Function composition is denoted $f \circ g$ and is equal to $\lambda x.f(g(x))$. It is assumed to have a lower precedence than application, so that $f \circ g(x) = f(g(x))$.

A program is a set of mutually recursive definitions, where a definition has the form $x = e$ with identifier x and expression e.

2.3 Algebra of Programs

To perform program transformations, we need an equational theory for Crystal that provides all the algebraic manipulations. The theory consists of the following:

- Definitions are treated as equations (with a simple left-hand side).
- The equational theory of the simply-typed lambda-calculus with constants, and with α-, β-, and η- conversion, substitution, and normalization rules.
- Other rules for dealing with functional language constructs such as composition distributing over λ-abstraction and conditional.
- The usual equalities involving Booleans and numbers.

2.4 Parallel Interpretation of Crystal Programs

The parallel interpretation of index domains is as spacetime locations (in memory, processor, or iteration steps). Parallel computation is then specified by defining a functions over index domains, called *data fields*. Data fields, therefore, unify the concept of arrays and other distributed data structures with functions. The reference patterns in the body of the functions determine the data dependencies between data elements. The recursive definitions are computed inductively, starting at the boundary conditions, rather than recursively as in a standard interpreter for a functional language.

The task of the Crystal compiler is to map parts of the data fields onto the nodes of a parallel machine, using the *data dependencies* between the elements of the mutually defined data fields to determine a partition that will minimize interprocessor communication [7, 5].

3 Domain Morphisms and Program Transformation

Let $\phi[x]$ indicate an expression possibly containing the identifier x. Suppose we are given a data field definition $a = \lambda(x){:}D.\phi_1[a, x]$ with codomain V. Operationally, it describes some computation over D with communication pattern given by its data dependencies. In many situations involving uniform recurrence equations, the kind of transformation that is done on the definition of a to optimize, for example, the communication, can be represented by a simple domain morphism g from D to E. For many optimizations g will be uniform, linear or affine.

Suppose $g = \lambda(x){:}D.\phi_2[x]$ is given with the inverse $h = \lambda(x){:}E.\phi_3[x]$, i.e., $h \circ g = \mathsf{id}_D$ and $g \circ h = \mathsf{id}_E$. Now we can algebraically derive the definition of a new data field, say $\tilde{a}$, which satisfies the following diagram:

$$\begin{tikzcd} D \arrow[r, "a"] \arrow[d, shift right, "g"'] & V \\ E \arrow[u, shift right, "h"'] \arrow[ur, "\tilde{a}"'] & \end{tikzcd}$$

The full derivation proceeds as follows:

$$\begin{aligned}
\tilde{a} &= a \circ h && \{\text{by definition}\} \\
&= \lambda(x){:}E.a \circ h(x) && \{\eta\text{-expand on the argument for } h\} \\
&= \lambda(x){:}E.a(h(x)) && \{\text{unfold composition}\} \\
&= \lambda(x){:}E.a(\phi_3[x]) && \{\text{unfold } h\} \\
&= \lambda(x){:}E.\phi_1[a, \phi_3[x]] && \{\text{unfold } a\} \\
&= \lambda(x){:}E.\phi_1[\tilde{a} \circ g, \phi_3[x]] && \{\text{substitute } a \text{ with } \tilde{a} \circ g\} \\
&= \lambda(x){:}E.\phi_1[\lambda(y){:}D.\tilde{a} \circ g(y), \phi_3[x]] && \{\eta\text{-expand on the argument for } g\} \\
&= \lambda(x){:}E.\phi_1[\lambda(y){:}D.\tilde{a}(g(y)), \phi_3[x]] && \{\text{unfold composition}\} \\
&= \lambda(x){:}E.\phi_1[\lambda(y){:}D.\tilde{a}(\phi_2[y]), \phi_3[x]] && \{\text{unfold } g\}
\end{aligned}$$

Notice that the definition of $\tilde{a}$ no longer mentions a, g, or h. Even the occurrence of D in the body will be removed when further simplification is done.

Many examples can be found for program transformation of uniform recurrence equations where g is a linear or affine function [10]. Since the transformations are quite formal, elsewhere we describe a metalanguage for automating them [9, 11]

4 Piecewise Linear Program Transformation

We illustrate our program transformation method with a piecewise linear domain morphism applied to a transitive closure program. The point of this exercise is not on how good a schedule one can generate but rather on how to formally derive a new program algebraically, given a schedule.

4.1 Warshall's Algorithm

Given the adjacency matrix A for a graph G, the transitive closure of G is the matrix C, where $C(i, j)$ is true if there is a path from node i to node j in G. Otherwise, $C(i, j)$ is false. Figure 1 shows the Crystal program for Warshall's algorithm for computing transitive closures [8]. The index domain N specifies the size of the adjacency matrix. The data field A specifies the adjacency matrix, which is to be read in from input. The index domain D specifies the two-dimensional space and one-dimensional iteration index required for the full computation. The data field G carries out the actual computation. The data field C specifies the final answer. The correctness of program T1 is based on the following fact: There is a path from node i to node j by way of nodes no larger than k if and only if there is a path from i to k by way of nodes no larger than $k-1$ *and* there is a path from k to j by way of nodes no larger than $k-1$, *or* there is already a path from i to j by way of nodes no larger than $k-1$. It is clear that there is a path from node i to node j in G if and only if there is a path from i to j by way of nodes no larger than n.

$$
\begin{aligned}
N &= \mathsf{interval}(1, n) \\
A &= \lambda(i,j) : N^2.\langle\text{Boolean adjacency matrix, read from input}\rangle \\
D &= N \times N \times \mathsf{interval}(0, n) \\
G &= \lambda(i,j,k) : D.\left\{\begin{array}{l} k = 0 \rightarrow A(i,j) \\ k > 0 \rightarrow (G(i,k,k-1) \text{ and } G(k,j,k-1)) \text{ or } G(i,j,k-1) \end{array}\right\} \\
C &= \lambda(i,j) : N^2.G(i,j,n)
\end{aligned}
$$

Figure 1: Program T1, the initial abstract Crystal program for computing transitive closure.

A sequential implementation for Program T1 is a straightforward three-level nested loop, with k the outer loop and i and j the inner loops. However, devising an efficient parallel implementation involves additional issues such as reducing communication overhead as well as finding an efficient schedule. In this paper we do not address the issues of domain partitioning and communication aggregation. They can be done using the same general technique. So, without loss of generality, we interpret the index (i, j) as the location of a processor on a two dimensional mesh. Therefore, $G(i, j, k)$ means the value of G on processor (i, j) at iteration k.

4.2 Broadcast Removal

The flow of information can be viewed from two complementary ways: From the receiver's or from the sender's point of view.

From the receiver's view, in order to compute $G(i, j, k)$, the processor (i, j) at iteration k needs values from processors (i, k) and (k, j) from iteration $k - 1$ as well as from itself from iteration $k - 1$. For example, Figure 2 shows that to compute $G(3, 1, 2)$, processor $(3, 1)$ at iteration 2 needs values from processors $(3, 2)$ and $(2, 1)$ at iteration 1, as well as itself at iteration 1. Similarly for computing $G(3, 4, 2)$.

From the sender's view, for all (i, j, k) in D and $k \neq 0$, the value $G(k, j, k - 1)$ needs to be sent to the whole slice $G(*, j, k)$ (Figure 3(a)). Similarly, for all (i, j, k) in D and $k \neq 0$, $G(i, k, k - 1)$ needs to be sent to $G(i, *, k)$.

These data dependencies induce broadcastings, which must be linearized if the processors only have a fixed number of outgoing communication channels. This is done in a standard way by introducing two routing data fields I and J (Figure 4) which propagate G along the i and j coordinates using only nearest neighbour communication (Figure 3(b)).

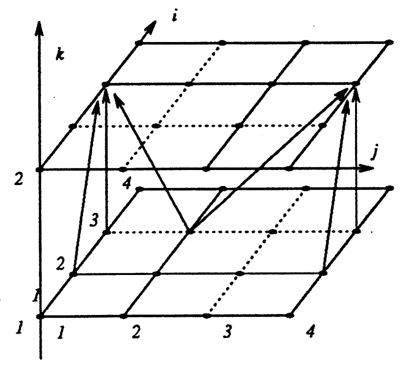

Figure 2: The data dependencies in the Program T1.

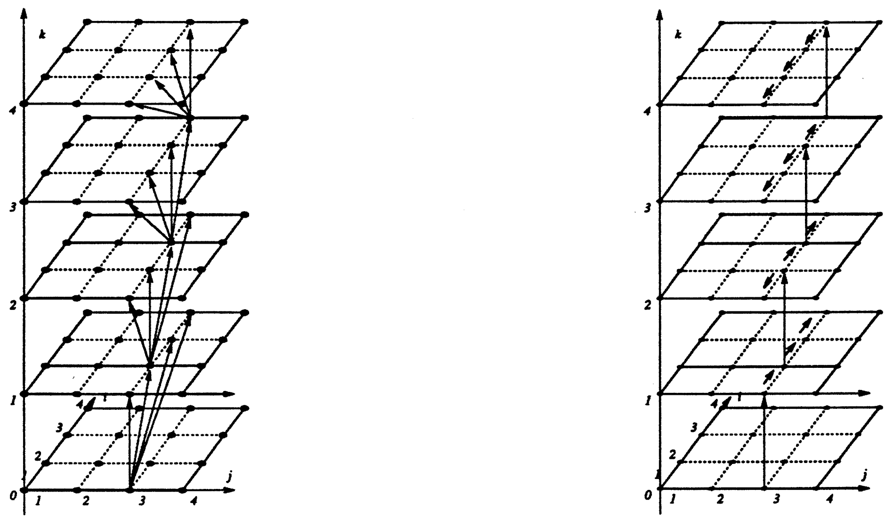

Figure 3: (a) $G(k,j,k-1)$ is broadcast to $G(*,j,k)$. (b) Linearize the broadcasting of $G(k,j,k-1)$ with the routing data field I.

It is easy to show that for all (i,j,k) in D and $k \neq 0$: $I(i,j,k) = G(k,j,k-1)$ and $J(i,j,k) = G(i,k,k-1)$.

The new program T2 is obtained by adding the definitions of I and J to program T1 and substituting $G(i,k,k-1)$ with $J(i,j,k)$ and substituting $G(k,j,k-1)$ with $I(i,j,k)$ in the definition of G. It has no time consuming broadcasts and uses only nearest-neighbor communication. The equivalence between Program T1 and T2 is established by the equivalence of the two definitions of G.

$$
\begin{aligned}
N &= \mathsf{interval}(1,n)\\
A &= \lambda(i,j) : N^2.\langle\text{adjacency matrix, read from input}\rangle\\
D &= N \times N \times \mathsf{interval}(0,n)\\
D' &= N \times N \times \mathsf{interval}(1,n)\\
I &= \lambda(i,j,k){:}D'.\left\{\begin{array}{l} i=k \rightarrow G(i,j,k-1)\\ i>k \rightarrow I(i-1,j,k)\\ i<k \rightarrow I(i+1,j,k)\end{array}\right\} \quad J = \lambda(i,j,k){:}D'.\left\{\begin{array}{l} j=k \rightarrow G(i,j,k-1)\\ j>k \rightarrow I(i,j-1,k)\\ j<k \rightarrow I(i,j+1,k)\end{array}\right\}\\
G &= \lambda(i,j,k){:}D.\left\{\begin{array}{l} k=0 \rightarrow A(i,j)\\ k>0 \rightarrow (J(i,j,k)\ \mathsf{and}\ I(i,j,k))\ \mathsf{or}\ G(i,j,k-1)\end{array}\right\}\\
C &= \lambda(i,j){:}N^2.G(i,j,n)
\end{aligned}
$$

Figure 4: Program T2: transitive closure without broadcasting.

4.3 Optimal Scheduling

A naive implementation of Program T2 using k as the iteration index will not be efficient. This is because at every iteration k, processor (i,j) needs $I(i,j,k)$, which must be propagated from the processor (k,j), as shown in Figure 3(b). This takes $|k-i|$ hops on a mesh-connected parallel architecture. It also needs $J(i,j,k)$, which must be propagated from processor (i,k) and takes $|k-j|$ hops. This implies that we will have a parallel program with time complexity of $O(n^2)$ even though we employ $O(n^2)$ processors. The inefficiency arises from using k as the iteration index, since processors are forced to wait for all others to finish their computation at iteration k before any can start computing on the next iteration $k+1$.

To optimize the efficiency, we should schedule processors to start computing as soon as all the necessary data are available. We will assume that it takes one time step to transfer data from a processor to its neighbors and that parallel computation proceeds in the "loosely synchronous" style [3, 6]. In each step, processors first communicate with their neighbors for the necessary data, then they do local computation using only local data. The global time index is used as the time stamp on when a value is computed.

Consider the following scheduling functions obtained by following the structure of the original code:

$$\tau_G = \lambda(i,j,k){:}D.\left\{\begin{array}{l} k = 0 \rightarrow 0 \\ k > 0 \rightarrow \max(\tau_G(i,j,k-1), \tau_I(i,j,k), \tau_J(i,j,k)) \end{array}\right\}$$

$$\tau_I = \lambda(i,j,k){:}D'.\left\{\begin{array}{l} i = k \rightarrow \tau_G(i,j,k-1) \\ i > k \rightarrow \tau_I(i-1,j,k)+1 \\ i < k \rightarrow \tau_I(i-1,j,k)+1 \end{array}\right\} \qquad \tau_J = \lambda(i,j,k){:}D'.\left\{\begin{array}{l} j = k \rightarrow \tau_G(i,j,k-1) \\ j > k \rightarrow \tau_J(i,j-1,k)+1 \\ j < k \rightarrow \tau_J(i,j+1,k)+1 \end{array}\right\}$$

It is not difficult to show, by induction, that the τ's give the earliest times when G, I, and J can be computed at the corresponding index points. The problem is that these recursively defined scheduling functions have no simple closed form representations, so that computing the these schedules would be as difficult as evaluating the original functions.

4.4 Scheduling with a Constant-Time Function

So, we try for a simply defined schedule that is consistent with the data dependencies in the program. We also prefer a single scheduling function for all the data fields so that the control structure of the parallel computation can be simplified.

A function $\tau : D \rightarrow \mathsf{Nat}$ is said to be a *legal schedule* for a set of mutually recursive data fields $F_i : D \rightarrow V$ if for all F_i, F_j and for all x, y in D, if $F_i(x)$ depends on $F_j(y)$ in the data field definitions, then

1. $\tau(x) \geq \tau(y)$, if x and y are on the same processor, or
2. $\tau(x) \geq \tau(y) + d$, if x and y reside on different processors d hops away.

With this definition as guide, we propose the following scheduling function:

$$\tau = \lambda(i,j,k){:}D.\left\{\begin{array}{l} k = 0 \rightarrow 0 \\ k > 0 \rightarrow 3k + |k-i| + |k-j| \end{array}\right\}$$

Lemma 1 *The function τ is a legal schedule for program* T2.

Proof: We need to verify that for all pairs of dependencies in Program T2, τ satisfies the requirements for being legal. We have the following dependencies in Program T2:

1. From the definition of G, (i,j,k) depends on (i,j,k) when $k \neq 0$. We have $\tau(i,j,k) = \tau(i,j,k)$.
2. From the definition of I, J and G, (i,j,k) depends on $(i,j,k-1)$ when $k \neq 0$ and $(i = k$ or $j = k)$. We have $\tau(i,j,k) = 3k + |k-i| + |k-j| > 3(k-1) + |k-1-i| + |k-1-j| = \tau(i,j,k-1)$.
3. From the definition of I, (i,j,k) depends on $(i-1,j,k)$ when $k \neq 0$ and $i > k$. We have $\tau(i,j,k) = 3k + i - k + |k-j| \geq 3k + i - 1 - k + |k-j| = \tau(i-1,j,k)+1$, if $i > k$.
4. From the definition of I, (i,j,k) depends on $(i+1,j,k)$ when $k \neq 0$ and $i < k$. We have $\tau(i,j,k) = 3k + k - i + |k-j| \geq 3k + k - i - 1 + |k-j| = \tau(i+1,j,k)+1$, if $k > i$.

5. From the definition of J, (i,j,k) depends on $(i,j-1,k)$ when $k \neq 0$ and $j > k$. We have $\tau(i,j,k) = 3k + |k-i| + j - k \geq 3k + |k-i| + j - 1 - k = \tau(i,j-1,k)+1$, if $j > k$.
6. From the definition of J, (i,j,k) depends on $(i,j+1,k)$ when $k \neq 0$ and $j < k$. We have $\tau(i,j,k) = 3k + |k-i| + k - j \geq 3k + |k-i| + k - j - 1 = \tau(i,j+1,k)+1$, if $j < k$.

□

4.5 Piecewise Linear Domain Morphisms

To derive a new program utilizing the formal transformation technique and the scheduling function τ, we need a pair of bijective domain morphisms. While τ is a legal constant-time schedule, it contains the non-linear absolute value operator $|\cdot|$. We can replace it with linear operators and conditional:

$$\tau = \lambda(i,j,k){:}D.\left\{\begin{array}{l} i \geq k \text{ and } j \geq k \rightarrow k+i+j \\ i \geq k \text{ and } j < k \rightarrow 3k+i-j \\ i < k \text{ and } j \geq k \rightarrow 3k-i+j \\ i < k \text{ and } j < k \rightarrow 5k-i-j \end{array}\right\}$$

The scheduling function τ is a piecewise linear function with different schedules in each of the four subdomains of D defined by the two planes $i = k$ and $j = k$. The four subdomains are

$$\begin{aligned} D_1 &= D|(\lambda(i,j,k).i \geq k \text{ and } j \geq k) \\ D_2 &= D|(\lambda(i,j,k).i \geq k \text{ and } j < k) \\ D_3 &= D|(\lambda(i,j,k).i < k \text{ and } j \geq k) \\ D_4 &= D|(\lambda(i,j,k).i < k \text{ and } j < k) \end{aligned}$$

On a cube of side n, the two planes cut the cube diagonally in different directions, each containing a bottom and a top edge. Then, D_1 is the pyramid with base the base of the cube and the point at (n,n,n), D_4 is an upside down pyramid, with base the top of the cube and point at $(0,0,0)$. D_2 and D_3 are tetrahedrons nestled in the space between the first two, on either side of the cube.

Using τ to generate the time index and using the identity mapping on the processor indices (i,j), we define the scheduling domain morphism $g : D \rightarrow E$ as below:

$$g = \lambda(i,j,k){:}D.\left\{\begin{array}{l} (i,j,k) \in D_1 \rightarrow (i,j,k+i+j) \\ (i,j,k) \in D_2 \rightarrow (i,j,3k+i-j) \\ (i,j,k) \in D_3 \rightarrow (i,j,3k-i+j) \\ (i,j,k) \in D_4 \rightarrow (i,j,5k-i-j) \end{array}\right\}{:}E$$

where $E = N \times N \times \text{interval}(0,5n)$.

Clearly, the four domains D_1, D_2, D_3, and D_4 are mutually disjoint. Let g_i be the component linear domain morphisms of g:

$$\begin{aligned} g_1 &= \lambda(i,j,k){:}D_1.(i,j,k+i+j) : E_1 \\ g_2 &= \lambda(i,j,k){:}D_2.(i,j,3k+i-j) : E_2 \\ g_3 &= \lambda(i,j,k){:}D_3.(i,j,3k-i+j) : E_3 \\ g_4 &= \lambda(i,j,k){:}D_4.(i,j,5k-i-j) : E_4 \end{aligned}$$

where E_i are the images of g_i, respectively.

Lemma 2 *The domain morphisms g_i are injective.*

Proof: We prove that g_1 is injective. The proofs for the rest are very similar. We show that if $(i,j,k) \neq (i',j',k')$ then $g_1(i,j,k) \neq g_1(i',j',k')$.

If $(i,j,k) \neq (i',j',k')$, then at least one of the followings must be true: (1) $i \neq i'$, (2) $j \neq j'$, or (3) $k \neq k'$. If (1) or (2) holds, then $g_1(i,j,k) = (i,j,k+i+j) \neq (i',j',k'+i'+j')$. If (1) and (2) do not hold, i.e., $i = i'$ and $j = j'$, then $k \neq k'$. Therefore, $g(i,j,k) = (i,j,k+i+j) \neq (i',j',k'+i'+j') = g(i',j',k')$ since $k+i+j \neq k'+i'+j'$. Hence, g_1 is injective. $\square$

Lemma 3 *The domain morphism g is injective.*

Proof: We show that if $(i,j,k) \neq (i',j',k')$, then $g(i,j,k) \neq g(i',j',k')$. This is clearly true when $i \neq i'$ or $j \neq j'$. When $i = i'$, $j = j'$, and $k \neq k'$:

1. If (i,j,k) and (i,j,k') are in the same subdomain, then $g(i,j,k) \neq g(i,j,k')$ by Lemma 2.
2. If (i,j,k) and (i,j,k') are in two different subdomains, say D_u and D_v, $u \neq v, 1 \leq u,v \leq 4$. There are six possibilities:
   (a) If (i,j,k) is in D_1 and (i,j,k') is in D_2, then $k \leq j < k'$. Since j and k are strictly less than k', $2j + k < 3k' \Rightarrow k+i+j < 3k'+i-j \Rightarrow g(i,j,k) = (i,j,k+i+j) \neq g(i,j,k') = (i,j,3k'+i-j)$.
   (b) If (i,j,k) is in D_1 and (i,j,k') is in D_3, then $k \leq i < k'$. Since i and k are strictly less than k', $2i + k < 3k' \Rightarrow k+i+j < 3k'-i+j \Rightarrow g(i,j,k) = (i,j,k+i+j) \neq g(i,j,k') = (i,j,3k'-i+j)$.
   (c) If (i,j,k) is in D_1 and (i,j,k') is in D_4, then $k \leq i,j < k'$. Since i,j and k are strictly less than k', $2i+2j+k < 5k' \Rightarrow k+i+j < 5k'-i-j \Rightarrow g(i,j,k) = (i,j,k+i+j) \neq g(i,j,k') = (i,j,5k'-i-j)$.
   (d) If (i,j,k) is in $D_2, (i,j,k')$ is in D_3, then we know $k \leq i < k'$ and $k' \leq j < k$, which is impossible.
   (e) If (i,j,k) is in $D_2, (i,j,k')$ is in D_4, then we know $k \leq i < k'$. Since i and k are strictly less than k', $3k + 2i < 5k' \Rightarrow 3k+i-j < 5k'-i-j \Rightarrow g(i,j,k) = (i,j,3k+i-j) \neq g(i,j,k') = (i,j,5k'-i-j)$.
   (f) If (i,j,k) is in $D_3, (i,j,k')$ is in D_4, then we know $k \leq j < k'$. Since j and k are strictly less than k', $3k + 2j < 5k' \Rightarrow 3k-i+j < 5k'-i-j \Rightarrow g(i,j,k) = (i,j,3k-i+j) \neq g(i,j,k') = (i,j,5k'-i-j)$.

$\square$

Lemma 4 *The domains E_i are mutually disjoint.*

Proof: This is a direct result of Lemma 3. $\square$

For each of the linear domain morphisms, we can derive their inverses by solving the corresponding simple simultaneous linear system of equations:

$$\begin{aligned} h_1 &= \lambda(x,y,t){:}E_1.(x,y,t-x-y) \\ h_2 &= \lambda(x,y,t){:}E_2.(x,y,(t-x+y)/3) \\ h_3 &= \lambda(x,y,t){:}E_3.(x,y,(t+x-y)/3) \\ h_4 &= \lambda(x,y,t){:}E_4.(x,y,(t+x+y)/5) \end{aligned}$$

Lemma 5 *The domain morphisms h_i are all injective.*

Proof: The proof is similar to Lemma 2's. $\square$

Lemma 6 *The domain morphisms g_i's and h_i's are bijective pairs.*

Proof: We need to show that $g_i \circ h_i = \mathsf{id}_{D_i}$ and $h_i \circ g_i = \mathsf{id}_{E_i}$ for each i. For $i = 2$, $h_2(g_2(i,j,k)) = h_2(i,j,3k+i-j) = (i,j,((3k+i-j)-i+j)/3) = (i,j,k)$ and $g_2(h_2(x,y,t)) = g_2(x,y,(t-x+y)/3) = (x,y,(t-x+y)+x-y) = (x,y,t)$. $\square$

We can now define the inverse of g:

$$h \;=\; \lambda(x,y,t){:}E.\left\{\begin{array}{l} h_1(x,y,t) \in D_1 \rightarrow h_1(x,y,t) \\ h_2(x,y,t) \in D_2 \rightarrow h_2(x,y,t) \\ h_3(x,y,t) \in D_3 \rightarrow h_3(x,y,t) \\ h_4(x,y,t) \in D_4 \rightarrow h_4(x,y,t) \end{array}\right\}{:}D,$$

where $E = E_1 + E_2 + E_3 + E_4$ is the image of g. And conclude from the previous Lemmas that

Lemma 7 *The domain morphisms g and h are bijective.*

4.6 Deriving the Efficiently Scheduled Program

Now we can utilize the equational derivation technique to algebraically derive the new program with the optimized explicit schedule. The new program consists of the new data fields $\tilde{I}, \tilde{J}, \tilde{G} : E \rightarrow V$ that make the diagram below commute:

$$\begin{tikzcd} D \arrow[r, "I{,}J{,}G"] \arrow[d, shift right, "g"'] & V \\ E \arrow[u, shift right, "h"'] \arrow[ur, "\tilde{I}{,}\tilde{J}{,}\tilde{G}"'] & \end{tikzcd}$$

The derivation for $\tilde{G}$:

$$\begin{array}{rcl} \tilde{G} & = & G \circ h \\ & = & \lambda(x,y,t){:}E.G(h(x,y,t)) \\ & = & \lambda(x,y,t){:}E.G(\left\{\begin{array}{l} h_1(x,y,t) \in D_1 \rightarrow h_1(x,y,t) \\ h_2(x,y,t) \in D_2 \rightarrow h_2(x,y,t) \\ h_3(x,y,t) \in D_3 \rightarrow h_3(x,y,t) \\ h_4(x,y,t) \in D_4 \rightarrow h_4(x,y,t) \end{array}\right\}) \\ & = & \lambda(x,y,t){:}E.\left\{\begin{array}{l} h_1(x,y,t) \in D_1 \rightarrow G(h_1(x,y,t)) \\ h_2(x,y,t) \in D_2 \rightarrow G(h_2(x,y,t)) \\ h_3(x,y,t) \in D_3 \rightarrow G(h_3(x,y,t)) \\ h_4(x,y,t) \in D_4 \rightarrow G(h_4(x,y,t)) \end{array}\right\} \\ & = & \lambda(x,y,t){:}E.\left\{\begin{array}{l} (x,y,t-x-y) \in D_1 \rightarrow G(x,y,t-x-y) \\ (x,y,(t-x+y)/3) \in D_2 \rightarrow G(x,y,(t-x+y)/3) \\ (x,y,(t+x-y)/3) \in D_3 \rightarrow G(x,y,(t+x-y)/3) \\ (x,y,(t+x+y)/5) \in D_4 \rightarrow G(x,y,(t+x+y)/5) \end{array}\right\} \end{array}$$

$$
\begin{aligned}
= \ & \lambda(x,y,t){:}E.\\
& \left\{\begin{array}{l}
(x,y,t-x-y) \in D_1 \rightarrow \\
\quad \left\{\begin{array}{l}
t-x-y=0 \rightarrow E(x,y)\\
\text{else}\rightarrow (J(x,y,t-x-y) \text{ and } I(x,y,t-x-y))\\
\qquad \text{or } G(x,y,t-x-y-1)
\end{array}\right\}\\
(x,y,(t-x+y)/3) \in D_2 \rightarrow \\
\quad \left\{\begin{array}{l}
(t-x+y)/3=0 \rightarrow E(x,y)\\
\text{else}\rightarrow (J(x,y,(t-x+y)/3) \text{ and } I(x,y,(t-x+y)/3))\\
\qquad \text{or } G(x,y,(t-x+y)/3-1)
\end{array}\right\}
\end{array}\right\}\\
& \left\{\begin{array}{l}
(x,y,(t+x-y)/3) \in D_3 \rightarrow \\
\quad \left\{\begin{array}{l}
(t+x-y)/3=0 \rightarrow E(x,y)\\
\text{else}\rightarrow (J(x,y,(t+x-y)/3) \text{ and } I(x,y,(t+x-y)/3))\\
\qquad \text{or } G(x,y,(t+x-y)/3-1)
\end{array}\right\}\\
(x,y,(t+x+y)/5) \in D_4 \rightarrow \\
\quad \left\{\begin{array}{l}
(t+x+y)/5=0 \rightarrow E(x,y)\\
\text{else}\rightarrow (J(x,y,(t+x+y)/5) \text{ and } I(x,y,(t+x+y)/5))\\
\qquad \text{or } G(x,y,(t+x+y)/5-1)
\end{array}\right\}
\end{array}\right\}\\
= \ & \lambda(x,y,t){:}E.\\
& \left\{\begin{array}{l}
(x,y,t-x-y) \in D_1 \rightarrow \\
\quad \left\{\begin{array}{l}
(t-x-y)=0 \rightarrow E(x,y)\\
\text{else}\rightarrow (\tilde{J}(g(x,y,t-x-y)) \ \text{ and } \tilde{I}(g(x,y,t-x-y))) \text{ or}\\
\quad \tilde{G}(g(x,y,t-x-y-1))
\end{array}\right\}\\
(x,y,(t-x+y)/3) \in D_2 \rightarrow \\
\quad \left\{\begin{array}{l}
(t-x+y)/3=0 \rightarrow E(x,y)\\
\text{else}\rightarrow (\tilde{J}(g(x,y,(t-x+y)/3)) \ \text{ and } \tilde{I}(g(x,y,(t-x+y)/3))) \text{ or}\\
\quad \tilde{G}(g(x,y,(t-x+y)/3-1))
\end{array}\right\}
\end{array}\right\}\\
& \left\{\begin{array}{l}
(x,y,(t+x-y)/3) \in D_3 \rightarrow \\
\quad \left\{\begin{array}{l}
(t+x-y)/3=0 \rightarrow E(x,y)\\
\text{else}\rightarrow (\tilde{J}(g(x,y,(t+x-y)/3)) \ \text{ and } \tilde{I}(g(x,y,(t+x-y)/3))) \text{ or}\\
\quad \tilde{G}(g(x,y,(t+x-y)/3-1))
\end{array}\right\}\\
(x,y,(t+x+y)/5) \in D_4 \rightarrow \\
\quad \left\{\begin{array}{l}
(t+x+y)/5=0 \rightarrow E(x,y)\\
\text{else}\rightarrow (\tilde{J}(g(x,y,(t+x+y)/5)) \ \text{ and } \tilde{I}(g(x,y,(t+x+y)/5))) \text{ or}\\
\quad \tilde{G}(g(x,y,(t+x+y)/5-1))
\end{array}\right\}
\end{array}\right\}
\end{aligned}
$$

$$
\begin{aligned}
= \ & \lambda(x,y,t){:}E. \\
& \left\{ \begin{array}{l}
(x,y,t-x-y) \in D_1 \rightarrow \\
\quad \left\{ \begin{array}{l}
t = x+y \rightarrow E(x,y) \\
\text{else} \rightarrow \tilde{J}(x,y,t) \text{ and } \tilde{I}(x,y,t) \text{ or} \\
\quad \left\{ \begin{array}{l}
(x,y,t-x-y-1) \in D_1 \rightarrow \tilde{G}(x,y,t-1) \\
(x,y,t-x-y-1) \in D_2 \rightarrow \tilde{G}(x,y,3t-2x-4y-3) \\
(x,y,t-x-y-1) \in D_3 \rightarrow \tilde{G}(x,y,3t-4x-2y-3) \\
(x,y,t-x-y-1) \in D_4 \rightarrow \tilde{G}(x,y,5t-4x-4y-5)
\end{array} \right\}
\end{array} \right\} \\
(x,y,(t-x+y)/3) \in D_2 \rightarrow \\
\quad \left\{ \begin{array}{l}
t+y = x \rightarrow E(x,y) \\
\text{else} \rightarrow \tilde{J}(x,y,t) \text{ and } \tilde{I}(x,y,t) \text{ or} \\
\quad \left\{ \begin{array}{l}
(x,y,(t-x+y)/3-1) \in D_1 \rightarrow \tilde{G}(x,y,(t+2x+4y)/3-1) \\
(x,y,(t-x+y)/3-1) \in D_2 \rightarrow \tilde{G}(x,y,t-3) \\
(x,y,(t-x+y)/3-1) \in D_3 \rightarrow \tilde{G}(x,y,t-2x+2y-3) \\
(x,y,(t-x+y)/3-1) \in D_4 \rightarrow \tilde{G}(x,y,(5k-8x+2y)/3-5)
\end{array} \right\}
\end{array} \right\}
\end{array} \right\}
\end{aligned}
$$

$$
\left\{ \begin{array}{l}
(x,y,(t+x-y)/3) \in D_3 \rightarrow \\
\quad \left\{ \begin{array}{l}
t+x = y \rightarrow E(x,y) \\
\text{else} \rightarrow \tilde{J}(x,y,t) \text{ and } \tilde{I}(x,y,t) \text{ or} \\
\quad \left\{ \begin{array}{l}
(x,y,(t+x-y)/3-1) \in D_1 \rightarrow \tilde{G}(x,y,(t+4x+2y)/3-3) \\
(x,y,(t+x-y)/3-1) \in D_2 \rightarrow \tilde{G}(x,y,t+2x-2y-3) \\
(x,y,(t+x-y)/3-1) \in D_3 \rightarrow \tilde{G}(x,y,t-3) \\
(x,y,(t+x-y)/3-1) \in D_4 \rightarrow \tilde{G}(x,y,(5k+2x-8y)/3-5)
\end{array} \right\}
\end{array} \right\} \\
(x,y,(t+x+y)/5) \in D_4 \rightarrow \\
\quad \left\{ \begin{array}{l}
t+x+y = 0 \rightarrow E(x,y) \\
\text{else} \rightarrow \tilde{J}(x,y,t) \text{ and } \tilde{I}(x,y,t) \text{ or} \\
\quad \left\{ \begin{array}{l}
(x,y,(t+x+y)/5-1) \in D_1 \rightarrow \tilde{G}(x,y,(t+6x+6y)/5-1) \\
(x,y,(t+x+y)/5-1) \in D_2 \rightarrow \tilde{G}(x,y,(t+8x-2y)/5-3) \\
(x,y,(t+x+y)/5-1) \in D_3 \rightarrow \tilde{G}(x,y,(t-2x+8y)/5-3) \\
(x,y,(t+x+y)/5-1) \in D_4 \rightarrow \tilde{G}(x,y,t-5)
\end{array} \right\}
\end{array} \right\}
\end{array} \right\}
$$

The derivations for I and J are identical.

5 Conclusion

Using a transitive closure algorithm, we have illustrated our program derivation method where an efficient schedule required a partitioning of the original index domain by planes. This meant that the domain morphism would have to be a piecewise linear function. Before we could apply the algebraic transformation we needed to have an inverse. Constructing it took the most effort, but once done, the rest of the transformation proceeded along the same general technique.

The power of our method comes from the rich equational theory of functions. The application of the approach is not limited to regular problems. With suitable extensions, more sophisticated transformations can be done where the domain morphisms can be time dependent, to reflect dynamically changing computational structures.