A Unifying Framework for Iteration Reordering Transformations

Wayne Kelly and William Pugh
Department of Computer Science
University of Maryland, College Park, MD 20742
{wak,pugh}@cs.umd.edu

February 27, 1995

Abstract

We present a framework for unifying iteration reordering transformations such as loop interchange, loop distribution, skewing, tiling, index set splitting and statement reordering. The framework is based on the idea that a transformation can be represented as a mapping from the original iteration space to a new iteration space. The framework is designed to provide a uniform way to represent and reason about transformations. We also provide algorithms to test the legality of mappings, and to generate optimized code for mappings.

1 Introduction

Optimizing compilers reorder iterations of statements to improve instruction scheduling, register use, and cache utilization, and to expose parallelism. Many different reordering transformations have been developed and studied, such as loop interchange, loop distribution, skewing, tiling, index set splitting and statement reordering [AK87, Wol89b, Wol90, CK92]. Each of these transformations has its own special legality checks and transformation rules. These checks and rules make it hard to analyze or predict the effects of compositions of these transformations, without performing the transformations and analyzing the resulting code.

Unimodular transformations [Ban90, WL91] go some way towards solving this problem. Unimodular transformations is a unified framework that is able to describe any transformation that can be obtained by composing loop interchange, loop skewing, and loop reversal. Such a transformation is described by a unimodular linear mapping from the original iteration space to a new iteration space. For example, loop interchange in a doubly nested loop maps iteration $[i,j]$ to iteration $[j,i]$. This transformation can be described using a unimodular matrix:

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

Unfortunately, unimodular transformations are limited in two ways: they can only be applied to perfectly nested loops and all statements in the loop nest are transformed in the same way. They therefore cannot represent some important transformations such as loop fusion, loop distribution and statement reordering.

1.1 Mappings

A mapping has the following general form:

$$\{\ [i^1,\ldots,i^m] \rightarrow [f^1,\ldots,f^n]\}$$

where:

- The iteration variables $i^1,\ldots,i^m$ represent the loops nested around the statement(s).
- The f^j's are functions of the iteration variables.

This mapping represents the fact that iteration $[i^1,\ldots,i^m]$ in the original iteration space is mapped to iteration $[f^1,\ldots,f^n]$ in the new iteration space. The implicit assumption is that the iterations in the new iteration space will be executed in lexicographic order. So, by specifying a mapping, we are really specifying a reordering of the iterations.
For example the above unimodular transformation would be represented by the mapping $\{\ [i,j] \rightarrow [j,i]\ \}$
In the case of unimodular transformations:

- All statements are mapped by the same mapping.
- The f^j's are linear functions.
- The mapping is invertable and unimodular.

- The dimensionality of the old and new iteration spaces are the same (i.e., $m = n$).

In our framework we generalize unimodular transformations in the following ways:

- We specify a separate mapping for each statement
- We allow the f^j to be psuedo affine rather than just linear. Affine means that they can include a (possibly symbolic) constant term. Pseudo affine means that they can also involve integer division and modulo operations (provided the denominator is a known integer constant).
- We require the mapping to be invertable, but not necessarily unimodular.
- We allow the dimensionality of the old and new iteration spaces to be different.
- We allow the mapping to be piecewise (as suggested by [Lu91]): we can specify a mapping T_p as $\bigcup_i T_{pi}|C_{pi}$ where the C_{pi}'s are disjoint sets which together contain all points in the iteration space of statement s_p. This mapping specifies that iteration i is mapped to the point $T_{pi}(i)$ iff $i \in C_{pi}$.

By generalizing in these ways, we can represent a much broader set of reordering transformations, including any transformation that can be obtained by some combination of loop interchange, loop reversal, loop skewing, statement reordering, loop distribution, loop fusion, loop blocking[1] (or tiling) [AK87] and index set splitting[1] [Ban79].

Figure 1 gives some interesting examples of mappings.

1.2 Overview

Our framework is designed to provide a uniform way to represent and reason about reordering transformations. The framework itself is not designed to decide which transformation should be applied. The framework should be used within some larger system, such as an interactive programming environment or an optimizing compiler. It is this surrounding system that is finally responsible for deciding which transformation should be applied. The framework does however provide some algorithms that would aid the surrounding system in its task. [KP94a] is an example of a surrounding system that uses our framework.

[1] Our current implementation cannot handle all cases of these transformations.

The rest of this paper is organized as follows. In Section 2 we describe how dependences and mappings are represented. In Section 3 we demonstrate that a large class of traditional transformations can be represented using mappings. In Section 4 we describe an algorithm that tests whether a mapping is legal. In Section 5 we describe our code generation algorithm. This algorithm takes a mapping and produces optimized code corresponding to the transformation represented by that mapping. By making use of the gist operation [PW92] we are able to produce code with a smaller number of conditionals and loop bounds than would otherwise be necessary. Finally we discuss related work, give our implementation status and state our conclusions.

2 Dependences and Mappings

Most of the previous work on program transformations uses data dependence directions and distances to summarize dependences between array references. These abstractions are sufficient for simple transformations such as unimodular transformations, but they are not precise enough to determine the legality of loop fusion and a number of other transformations without actually applying the transformation and re-evaluating dependences. Since our framework includes loop fusion, they are not sufficient for our purposes either. We evaluate and represent dependences exactly using affine constraints over integer variables. We use the Omega Library [Pug92, PW92] to manipulate and simplify these constraints.

The following is a brief description of integer tuple relations and dependence relations.

2.1 Integer tuple relations and sets

An integer k-tuple is simply a point in $\mathcal{Z}^k$. A *tuple relation* is a mapping from tuples to tuples. A single tuple may be mapped to zero, one or more tuples. A relation can be thought of as a set of pairs, each pair consisting of an input tuple and its associated output tuple. All the relations we consider map from k-tuples to k'-tuples for some fixed k and k'. The relations may involve free variables such as n in the following example: $\{ [i] \rightarrow [i{+}1] \mid 1 \leq i < n \}$. These free variables correspond to symbolic constants or parameters in the source program. We use Sym to represent the set of all symbolic constants. A relation is represented as the union of a set of *simple relations*: relations that can be described by the conjunction of a set of affine constraints. We can represent simple relations containing certain non-convex constraints

Code adapted from OLDA in Perfect club (TI)

Original code

```
      do 20 mp = 1, np
        do 20 mq = 1, mp
          do 20 mi = 1, morb
10           xrsiq(mi,mq)=xrsiq(mi,mq)+
     $                    xrspq((mp-1)*mp/2+mq)*v(mp,mi)
20           xrsiq(mi,mp)=xrsiq(mi,mp)+
     $                    xrspq((mp-1)*mp/2+mq)*v(mq,mi)
```

Mapping (to expose parallelism)

$$T_{10}: \{ [\ mp, \quad mq, \quad mi\ ] \quad \rightarrow [\ mi, \quad mq, \quad mp, \quad 0]\}$$
$$T_{20}: \{ [\ mp, \quad mq, \quad mi\ ] \quad \rightarrow [\ mi, \quad mp, \quad mq, \quad 1]\}$$

Transformed code

```
     do 20 mi = 1,morb   /* parallelizable */
       do 20 t2 = 1,np   /* parallelizable */
         do 10 t3 = 1,t2-1
10          xrsiq(mi,t2)=xrsiq(mi,t2) +
    $                   xrspq((t3-1)*t3/2+t2)*v(t3,mi)
         xrsiq(mi,t2)=xrsiq(mi,t2) +
    $                 xrspq((t2-1)*t2/2+t2)*v(t2,mi)
         xrsiq(mi,t2)=xrsiq(mi,t2) +
    $                 xrspq((t2-1)*t2/2+t2)*v(t2,mi)
         do 20 t3 = t2+1,np
20          xrsiq(mi,t2)=xrsiq(mi,t2) +
    $                   xrspq((t2-1)*t2/2+t3)*v(t3,mi)
```

Transformations required normally

- index set splitting
- loop distribution
- triangular loop interchange
- loop fusion

LU Decomposition without pivoting

Original code

```
    do 20 k = 1, n
      do 10 i = k+1, n
10      a(i,k) = a(i,k) / a(k,k)
        do 20 j = k+1, n
20        a(i,j) = a(i,j) - a(i,k) * a(k,j)
```

Mapping (for locality)

$$T_{10}: \{[k, i\ \ ] \rightarrow [64((k-1)\ div\ 64)+1, 64(i\ div\ 64), k, k, i]\}$$
$$T_{20}: \{[k, i, j] \rightarrow [64((k-1)\ div\ 64)+1, 64(i\ div\ 64), j, k, i]\}$$

Transformed code

```
     do 30 kB = 1, n-1, 64
       do 30 iB = kB-1, n, 64
         do 20 kj = kB, min(kB+63, n)
           do 10 k = kB, kj-1
             do 10 i = max(k+1,iB), min(iB+63,n)
10              a(i,kj)=a(i,kj)-a(i,k)*a(k,kj)
           do 20 i = max(iB,kj+1), min(iB+63,n)
20            a(i,kj)=a(i,kj)/a(kj,kj)
         do 30 kj = kB+64, n
           do 30 k = kB to kB+64
             do 30 i = max(k+1,iB), min(iB+63,n)
30              a(i,kj)=a(i,kj)-a(i,k)*a(k,kj)
```

Transformations required normally

- strip mining
- index set splitting
- loop distribution
- imperfectly nested triangular loop interchange

Code adapted from CHOSOL in the Perfect club (SD)

Original code

```
      do 30 i=2,n
10       sum(i) = 0.
        do 20 j=1,i-1
20         sum(i) = sum(i) + a(j,i)*b(j)
30       b(i) = b(i) - sum(i)
```

Mapping (to expose parallelism)

$$T_{10}: \{ [\ i\ ] \quad \rightarrow [\ 0, \quad i, \quad 0, \quad 0\ ]\ \}$$
$$T_{20}: \{ [\ i, \quad j\ ] \quad \rightarrow [\ 1, \quad j, \quad 0, \quad i\ ]\ \}$$
$$T_{30}: \{ [\ i\ ] \quad \rightarrow [\ 1, \quad i-1, \quad 1, \quad 0\ ]\ \}$$

Transformed code

```
     do 10 i = 2,n   /* parallelizable */
10      sum(i) = 0.
     do 30 t2 = 1, n-1
       do 20 i = t2+1,n   /* parallelizable */
20        sum(i) = sum(i) + a(t2,i)*b(t2)
30      b(t2+1) = b(t2+1) - sum(t2+1)
```

Transformations required normally

- loop distribution
- imperfectly nested triangular loop interchange

Banded SYR2K adapted from BLAS

Original code

```
      do 10 i = 1, n
       do 10 j = i, min(i+2*b-2,n)
        do 10 k = max(i-b+1,j-b+1,1),min(i+b-1,j+b-1,n)
10         C(i,j-i+1) = C(i,j-i+1) +
     $        alpha*A(k,i-k+b)*B(k,j-k+b) +
     $        alpha*A(k,j-k+b)*B(k,i-k+b)
```

Mapping (for locality and to expose parallelism)

$$T_{10}: \{ [\ i, \quad j, \quad k] \quad \rightarrow [\ j-i+1, \quad k-j, \quad k\ ]\ \}$$

Transformed code

```
      do 10 t1 = 1, min(n,2*b-1)   /* parallelizable */
        do 10 t2 = max(1-b,1-n), min(b-t1, n-t1)
          do 10 k = max(1,t1+t2), min(n+t2,n)   /* parallelizable */
10           C(-t1-t2+k+1,t1) = C(-t1-t2+k+1,t1) +
     $            alpha*A(k,-t1-t2+b+1)*B(k,-t2+b) +
     $            alpha*A(k,-t2+b)*B(k,-t1-t2+b+1)
```

Transformations required normally

- loop skewing
- triangular loop interchange

Figure 1: Example Codes, Mappings, and Resulting Transformations

operation	Description	Definition
F^{-1}	The inverse of F	$x \rightarrow y \in F^{-1} \Leftrightarrow y \rightarrow x \in F$
$S \cap T$	The intersection of S and T	$x \in S \cap T \Leftrightarrow x \in S \wedge x \in T$
$range(F)$	The range of F	$y \in range(F) \Leftrightarrow \exists x$ s.t. $x \rightarrow y \in F$
$restrict\_domain(F, S)$	F with domain restricted to S	$x \rightarrow y \in restrict\_domain(F, S) \Leftrightarrow x \rightarrow y \in F \wedge x \in S$
$\pi_{1,\ldots,v}\ S$	The projection of S onto $1, \ldots, v$	$x \in \pi_{1,\ldots,v}\ S \Leftrightarrow \lvert x \rvert = v \wedge \exists y$ s.t. $xy \in S$
$satisfiable(S)$	True if S is not empty	$satisfiable(S) \Leftrightarrow \exists x \in S$

Table 1: Operations on tuple sets and relations, where F and G are tuple relations and S and T are tuple sets

such as $\{\ [i] \rightarrow [i] \mid i \text{ even }\}$ by introducing wildcard variables (denoted by Greek letters):

$$\{\ [i] \rightarrow [i] \mid \exists \alpha \text{ s.t. } i = 2\alpha\ \}$$

Table 1 gives a brief description of the operations on integer tuple sets and relations that we use in this paper. See [Pug91] for a more thorough description.

2.2 Control dependence

We require that any if-then-else constructs be converted into guarded assignment statements or that they be treated as atomic statements. We also require that all loop bounds be affine functions of surrounding loop variables and symbolic constants. All control dependences can therefore be implicitly represented by describing the iteration space using a set of affine inequalities on the loop variables and symbolic constants.

2.3 Data dependence

From now on, when we refer to dependences, we will be implicitly referring to data dependences. We use tuple relations to represent dependences. If there is a dependence from $s_p[i]$ (i.e., iteration i of statement s_p) to $s_q[j]$ then the tuple relation d_{pq} representing the dependences from s_p to s_q will map $[i]$ to $[j]$ (i and j are tuples). We do not distinguish between different kinds of dependences (i.e., flow, output and anti-dependences) because they all impose ordering constraints on the iterations in the same way. It is possible to remove output and anti-dependences using techniques such as array and scalar expansion; we assume that this has already been done if it is desirable, and that the dependences have been updated. An alternative approach is to annotate the dependence information in such a way that certain dependences are ignored under the presumption that they can be removed if necessary.

2.4 gist, approx and hull

We make use of the **gist** operation that was originally developed in [PW92]. The **gist** operation is designed to operate on two relations, each of which is represented by a single conjunction of constraints. Intuitively, (**gist** p **given** q) is defined as the new information contained in p, given that we already know q. More formally, if $p \wedge q$ is satisfiable then (**gist** p **given** q) is a minimal conjunction such that

$$((\textbf{gist } p \textbf{ given } q) \wedge q) = (p \wedge q))$$

For example **gist** $1 \leq i \leq 10$ **given** $i \leq 5$ is $1 \leq i$. If $p \wedge q$ is not satisfiable then (**gist** p **given** q) is **False**.

The **approx** operation is designed to operate on a relation that is represented by a single conjunction of constraints. The **approx** operation is defined so that **approx**$(p) \supseteq p$ and **approx**(p) is convex. Within these constraints, **approx**(p) is made as tight as possible; if p is convex, **approx**$(p) = p$. The original set of constraints p may involve wildcard variables which may cause the region described by p to be non-convex. The **approx**(p) operation works by simplifying the constraints in p under the assumption that the wildcard variables can take on rational values. This allows us to eliminate all wildcard variables.

The **hull** operation is designed to operate on a set of relations, each of which is represented by a single conjunction of convex constraints. The **hull** of a set of relations $\{R_1, \ldots, R_m\}$ is a new relation R' that is a superset of each of the R_i relations. R must also have the property that it can be represented by a single conjunction of convex constraints. We try to make R' tight, but it is not necessarily what is commonly referred to as "The convex hull" of these relations.

2.5 Mappings

We associate a separate mapping with each statement. We therefore need a way to refer to the mappings of individual statements. We represent the mapping associated with statement s_p as

$$T_p : [i_p^1, \ldots, i_p^{m_p}] \rightarrow [f_p^1, \ldots, f_p^n] \mid C_p$$

The f_p^i expressions are called *mapping components*. For simplicity but without loss of generality we re-

Original_order (S, $[i_1, ..., i_k] \rightarrow [f_1, ..., f_a]$)
  case S **of**
    "for $i_{k+1} = \ldots$ to $\ldots$ do S_1":
      return Original_order (S_1, $[i_1, \ldots, i_k, i_{k+1}] \rightarrow [f_1, \ldots, f_a, i_{k+1}]$)
    "$S_1; S_2; \ldots ; S_m$":
      return $\bigcup_{p=1}^{m}$ **Original_order** (S_p, $[i_1, \ldots, i_k] \rightarrow [f_1, \ldots, f_a, p]$)
    "assignment #p":
      return $T_p : \{ [i_1, \ldots, i_k] \rightarrow [f_1, \ldots, f_a] \}$

Figure 2: Function to compute mapping that corresponds to the original execution order

quire that all of the mappings have n components. We refer to each of the positions $1, \ldots, n$ as *levels.*

We will use the term mapping to refer to both the mappings of individual statements and the set of mappings associated with all statements.

3 Traditional Transformations

In this section we demonstrate how mappings can be used to represent all transformations that can be obtained by applying any sequence of the traditional transformations listed in Section 1.

We will describe how to construct mappings to represent traditional transformations by describing how to modify mappings that correspond to the original execution order of programs. The mapping that corresponds to the original execution order of a program, can be constructed by a recursive descent of the abstract syntax tree (AST). Nodes in the AST have three forms: loops, statement lists and guarded assignment statements. The function **Original_order** (see Figure 2), when called with arguments of S and $[\,] \rightarrow [\,]$, returns a mapping for each of the assignment statements in S.

For example, the mapping that corresponds to the original execution order of the following program is:

$$T_1 : \{[i, j] \rightarrow [i, 1, j]\ \}$$
$$T_2 : \{[i, k] \rightarrow [i, 2, k]\ \}$$

```
for i = 1 to 10 do
    for j = 1 to 5 do
        s1(i,j) = 1
    for k = 0 to 100 do
        s2(i,j) = 2
```

When constructing these mappings we categorize mapping components as being either syntactic components (always an integer constant) or loop components (a function of the loop variables of that statement). **syntactic**(f_p^i) is a boolean function which is true iff f_p^i is a symbolic component (**loop**(f_p^i) is defined analogously). The common syntactic level (csl) of two statements s_p and s_q is defined as:

$$\mathrm{csl}(s_p, s_q) \equiv \min\{i-1 \mid \quad 1 \le i \wedge f_p^i \ne f_q^i \wedge \mathbf{syntactic}(f_p^i) \wedge \mathbf{syntactic}(f_q^i)\}$$

Intuitively, the common syntactic level of two statements is the deepest loop which surrounds both statements. Figure 3 describes how to construct mappings to represent traditional transformations by describing how to modify the mappings we have just described. Since these rules can be applied repeatedly, we can represent not only standard transformations but also any sequence of standard transformations.

4 Mapping Legality Test

In this section we describe an algorithm that tests whether a mapping is legal. A mapping is legal if the transformation it describes preserves the semantics of the original code. This is true if the new ordering of the iterations respects all of the dependences in the original code. In order words, if i is an iteration of statement s_p and j an iteration of statement s_q, and the dependence relation d_{pq} indicates that there is a dependence from i to j then $T_p(i)$ must be executed before $T_q(j)$. More formally,

$$\forall i, j, p, q, Sym\ i \rightarrow j \in d_{pq}\ \Rightarrow T_p(i) \prec T_q(j)$$

where Sym is the set of all symbolic constants in the equation, and $\prec$ is the lexicographic ordering operator. We verify this by using the Omega library to evaluate:

$$\neg \exists i, j, p, q, Sym\ \text{ s.t. } i \rightarrow j \in d_{pq} \wedge T_p(i) \succeq T_q(j)$$

We also require that the mapping be 1-1:

$$\forall p, q, i, j, Sym\ (p = q \wedge i = j) \Leftrightarrow T_p(i) = T_q(j)$$

Distribution Distribute loop at depth L over the statements D, with statement s_p going into ${r_p}^{\text{th}}$ loop.

Requirements: $\forall s_p, s_q \;\; s_p \in D \wedge s_q \in D \Rightarrow \mathbf{loop}(f_p^L) \wedge L \leq \mathrm{csl}(s_p, s_q)$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, r_p, f_p^L, \ldots, f_p^n]$

Statement Reordering Reorder statements D at level L so that new position of statement s_p is r_p.

Requirements: $\forall s_p, s_q \;\; s_p \in D \wedge s_q \in D \Rightarrow \mathbf{syntactic}(f_p^L) \wedge L \leq csl(s_p, s_q) + 1 \wedge$ $(L \leq csl(s_p, s_q) \Leftrightarrow r_p = r_q)$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, r_p, f_p^{(L+1)}, \ldots, f_p^n]$

Fusion Fuse the loops at level L for the statements D with statement s_p going into the ${r_p}^{\text{th}}$ loop.

Requirements: $\forall s_p, s_q \; s_p \in D \wedge s_q \in D \Rightarrow \mathbf{syntactic}(f_p^{(L-1)}) \wedge \mathbf{loop}(f_p^L) \wedge L - 2 \leq csl(s_p, s_q) + 2 \wedge$ $(L - 2 < csl(s_p, s_q) + 2 \Rightarrow r_p = r_q)$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-2)}, r_p, f_p^{(L)}, f_p^{(L-1)}, f_p^{(L+1)}, \ldots, f_p^n]$

Unimodular Transformation Apply a $k \times k$ unimodular transformation U to a perfectly nested loop containing statements D at depth $L \ldots L + k$. Note: Unimodular transformations include loop interchange, skewing and reversal [Ban90, WL91].

Requirements: $\forall i, s_p, s_q \;\; s_p \in D \wedge s_q \in D \wedge L \leq i \leq L + k \Rightarrow \mathbf{loop}(f_p^i) \wedge L + k \leq csl(s_p, s_q))$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, U[f_p^{(L)}, \ldots f_p^{(L+k)}]^\top, f_p^{(L+k+1)}, \ldots, f_p^n]$

Strip-mining Strip-mine the loop at level L for statements D with block size B

Requirements: $\forall s_p, s_q \;\; s_p \in D \wedge s_q \in D \Rightarrow \mathbf{loop}(f_p^L) \wedge L \leq csl(s_p, s_q)) \wedge B$ is a known integer constant

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, B(f_p^{(L)} \text{ div } B), f_p^{(L)}, \ldots, f_p^n]$

Index Set Splitting Split the index set of statements D using condition C

Requirements: C is affine expression of symbolic constants and indexes common to statements D.

Transformation: $\forall s_p \in D$, replace T_p by $(T_p \mid C) \;\cup\; (T_p \mid \neg C)$

Figure 3: Representing Traditional Transformations

so that the new program performs exactly the same set of computations as the original program. This can be similarly verified using the Omega library. We have also developed techniques [KP94b] to give a closed form characterization of the set of legal mappings.

5 Optimized Code Generation

In this section we describe an algorithm to generate efficient source code for a mapping. As an example, consider changing the KIJ version of Gaussian Elimination (without pivoting) into the IJK version (the version refers to the nesting of the loops around the inner most statement).

```
      do 20 k = 1, n
        do 10 i = k+1, n
10        a(i,k) = a(i,k)/a(k,k)
          do 20 j = k+1, n
20          a(i,j) = a(i,j)-a(k,j)*a(i,k)
```

Michael Wolfe notes that this transformation requires imperfect triangular loop interchange, distribution, and index set splitting [Wol91]. We can produce the IJK ordering using the following mapping:

$$
\begin{array}{llllllllllll}
T_{10}: & \{[& k, & i & &] & \rightarrow & [& i, & k, & 1, & 0 \;]\} \\
T_{20}: & \{[& k, & i, & j &] & \rightarrow & [& i, & j, & 0, & k \;]\}
\end{array}
$$

A naive code generation strategy would produce the following code:

```
      do 20 t0 = 2, n
        do 20 t1 = 1, n
          do 20 t2 = 0, 1
            do 20 t3 = 0, n
10            if (t1<t0.and.t2=1.and.t3=0)
     $          a(t0,t1) = a(t0,t1)/a(t1,t1)
              if (t3<t0.and.t3<t1.and.t2=0)
20   $          a(t0,t1) = a(t0,t1)-a(t3,t1)*a(t0,t3)
```

This is, of course, undesirable, because it results in about $6n^3$ unnecessary comparisions. This section explains how we produce the following more efficient code:

```
      do 40 i = 2,n
        a(i,1) = a(i,1)/a(1,1)
        do 30 t2 = 2,i-1
          do 20 k = 1,t2-1
20          a(i,t2) = a(i,t2)-a(k,t2)*a(i,k)
30        a(i,t2) = a(i,t2)/a(t2,t2)
        do 40 j = i,n
          do 40 k = 1,i-1
40          a(i,j) = a(i,j)-a(k,j)*a(i,k)
```

To simplify the discussion we do not consider piecewise mappings in this section (they can be handled by considering each piece of the mapping as a separate statement).

5.1 Old and new iteration spaces

We are currently able to transform programs that consist of guarded assignment statements surrounded by an arbitrary number of possibly imperfectly nested loops. For each statement s_p we combine information from the loop bounds and steps into a tuple set I_p that describes the original iteration space of that statement. If the mapping associated with statement s_p is T_p, then the statement's new iteration space I'_p is given by $restrict\_domain(T_p, I_p)$. This new iteration space is represented internally as a set of affine constraints and is the starting point for the rest of this section.

In the example above, the original iteration space is:

$$I_{10} : \{[k,i] \mid 1 \leq k \leq n \wedge k+1 \leq i \leq n\}$$
$$I_{20} : \{[k,i,j] \mid 1 \leq k \leq n \wedge k+1 \leq i \leq n \wedge k+1 \leq j \leq n\}$$

and the new iteration space is:

$$I'_{10} : \{[i,k,1,0] \mid 1 \leq k \leq n \wedge k+1 \leq i \leq n\}$$
$$I'_{20} : \{[i,j,0,k] \mid 1 \leq k \leq n \wedge k+1 \leq i \leq n \wedge k+1 \leq j \leq n\}$$

5.2 Code generation for a single level

We need to generate code that will iterate over all and only those points $[t_1, \ldots, t_n]$ in the new iteration space. The new code must execute the iterations in lexicographical order based on the new coordinate system. In the new code, each statement will be surrounded by n (possibly trivial) loops – one loop for each dimension of the new iteration space. The outermost loops iterates over values of the first index variable t_1, the next to outer most loops iterates over values of the second index variable t_2, and so on. This produces the desired result of having the iterations in the new iteration space execute in lexicographic order. Our code generation algorithm builds the transformed code recursively, level by level, starting at the first/outermost level (see Figure 4). In this sub-section we describe how code is generated for a single level L.

We introduce a new index variable t_L to be used for this level. For each statement s_p, we need to generate a set of constraints J_p that represents the values of t_L for which statement s_p should be executed. We cannot simply project I'_p onto t_L, because this will only give us absolute upper and lower bounds on t_L. Expressing the tightest possible upper and lower bounds on t_L may require using expressions that involve index variables from earlier levels. So we instead calculate the projection $\Pi_{1,\ldots,L}(I'_p)$. The result contains constraints on t_L, on index variables from earlier levels $(t_1, \ldots, t_{L-1})$, and on symbolic constants. Many of these constraints are redundant because the code we generated at earlier levels enforce them. We remove these redundant constraints and simplify others by making use of the information that is known about the values of index variables from earlier levels. So we have $J_p =$ gist $\Pi_{1,\ldots,L}(I'_p)$ given *known*.

The J_p sets for different statements may, in general, overlap. If J_p and J_q overlap over some range, then the t_L loop that iterates over that overlap range must contain both statements s_p and s_q (otherwise the iterations will not be executed in lexicographic order as required). In general, it is not possible to generate a loop containing more than one statement that iterates over exactly the J_p sets of the statements in the loop. The problem is that, in general, the J_p sets will have incompatible stride constraints. Stride constraints are constraints of the form $\exists\alpha$ s.t. $t_L = c\alpha + s$, where α is a wildcard variable. These sorts of constraints can appear in J_p if the coefficients of the mapping components are not ±1, or if the original program contains steps which are not ±1.

We solve this problem by removing all stride constraints from the J_p sets, and worry about adding them later. To remove the stride constraints, we use $AI_p = \text{approx}\ (J_p)$.

The constraints in AI_p now describe a continuous range of values for t_L. For purposes of explanation we define (but do not compute) $E = \bigcup_p AI_p$. Having removed the stride constraints, we can now put more than one statement into a loop, but there is still one problem remaining: If we were to generate a single t_L loop iterating over all points in E and containing all statements, then we would possibly still execute some statements with values of t_L not in their AI_p ranges. We could overcome this problem by adding guards around the elementary assignment statements. However, we prefer a more efficient solution.

We partition the range of values of the current index variable t_L such that within each partition, each statement is either not executed at all or is executed at every iteration within that partition. Instead of generating a single loop, we generate a sequence of loops, one for each partition.

We form these partitions as follows. We choose an arbitrary constraint c from some AI_q, that is not

```
GenerateCode ( T, I )
    for each (stmt)
        I'[stmt] = range ( restrict_domain ( T[stmt], I[stmt] ))
    return gen_recursive ( I', 1, True , True )

gen_recursive ( I', level, known, constraints )
    active = { s | ( I'[s] ∩ known ∩ constraints ) is satisfiable}
    for each (stmt) ∈ active
        J[stmt] = gist project ( I'[stmt], 1..level+1 ) given known
        AI[stmt] = approx ( J[stmt] )
    Hull = hull ( { AI[stmt] | stmt ∈ active } )
    if ( ∃ constraint c, stmt s s.t. c part of AI[s] ∧ s ∈ active ∧ c is not part of Hull )
        s1 = gen_recursive ( I', level, known, constraints::{not c} )
        s2 = gen_recursive ( I', level, known, constraints::{c} )
        if ( c is lower bound ) then return ( s1, ";", s2 )
        elseif ( c is upper bound ) then return ( s2, ";", s1 )
        else return ( "if" c "then" s1 "else" s2);
    else
        loop = constraints ∩ Hull ∩ greatest_common_step ( active ) )
        loop = gist loop given known
        loopCode = generate_loop ( level, loop )
        new_known = known ∩ loop
        if ( level < last_level ) then
            loopBody = gen_recursive ( I', level+1, new_known, True )
            return ( loopCode loopBody )
        else
            not_represented = gist I'[only active stmt] given new_known
            statement = generate_statement ( only active stmt )
            return ( loopCode "if" not_represented "then" statement)
```

Figure 4: Code Generation Algorithm

part of the hull of the AI_p's. For example, if

$$AI_1 = \{ [t_1] \mid 1 \leq t_1 \leq 15 \}$$

and

$$AI_2 = \{ [t_1] \mid 10 \leq t_1 \leq 20 \}$$

then we would choose either $t_1 \leq 15$ or $10 \leq t_1$. This constraint can be used to split the iteration space into two disjoint iteration spaces – the interval of the iteration space that satisfies the constraint c and the interval that satisfies the constraint $\neg c$. If c is an upper bound on the current index variable, then the interval of the iteration space that satisfies c will be entirely executed before the interval that does not satisfy c; if c is a lower bound, then the required order is reversed. If c does not involve the current index variable, then on any given execution, only one of the two intervals will be executed. In that case, instead of a loop, we generate an if-then-else construct, where the condition is c, the then clause iterates over those points satisfying c, and the else clause iterates over those points satisfying $\neg c$.

Since we always choose a constraint that is not part of the hull, we can be sure that both sections will contain some iterations. The interval satisfying $\neg c$ will not contain any iterations belonging to statement s. There are likely to be other statements s' whose iterations are entirely confined either to the interval satisfying c or the interval satisfying $\neg c$. We detect these cases by intersecting the iteration space of each statement with both c and $\neg c$ to determine if they are empty. If an intersection is non-empty then we say that the statement is active in that interval.

We repeat this splitting process recursively on both intervals. In each interval, we consider only those constraints that represent bounds on active statements. The recursion terminates when all of the constraints left to consider are part of the hull of the active statements.

Now that we know exactly which statements are active in each particular interval, we may be able to add some of the stride constraints that we removed earlier. Within a given interval, each statement s_p that is active in that interval may contribute a single stride constraint $\exists\beta$ s.t. $t_i = a_p\beta + b_p$ that was removed earlier (where a_p is an integer coefficient and b_p is an affine function of symbolic constants and outer level index variables).

We calculate the *greatest common step* of that interval as follows:

$$gcs = \begin{array}{l} gcd(\{a_p | s_p \text{ is active}\} \cup \\ \{gcd(b_q - b_p) | s_q \text{ is active} \wedge s_p \text{ is active}\}) \end{array}$$

The gcd (greatest common divisor) of an expression is defined to be the gcd of the coefficients in the expression.

We can now add to that interval, the constraint $\exists \beta$ s.t. $t_i = gcs\ \beta + b_p$, where s_p is an arbitrary active statement. We can safely add this constraint since if t_i satisfies the stride constraint of any active statement then it will also satisfy this constraint.

The greatest common step will be extracted from these constraints and used as the step for the loop corresponding to that interval. The step may not enforce all of the stride constraints on active statements, but we cannot add anything stronger without excluding required iterations. Any remaining stride constraints will be enforced at later levels.

For each interval, we generate a `do` loop to iterate over the appropriate values of t_L. Each of these intervals may contain a stride constraint of the form $\exists \beta$ s.t. $t_L = g\ \beta + c$. If this is the case we enforce the stride constraint by using a non-unit step in the loop. In order to do this however, we must ensure that the loop's lower bound satisfies the stride constraint.

The loop's lower bounds come from constraints of the form $lower \leq m\ t_L$. Such a constraint produces a lower bound expression of:

$$g \left\lceil \frac{lower - c\ m}{m\ g} \right\rceil + c$$

This expression can often be simplified once we know the values of g, m, c and $lower$. The loop's upper bounds come from constraints of the form $n\ t_L \leq upper$. It is sufficient to use $upper/n$ as an upper bound of the loop. The loop we generate at depth L, corresponding to interval i has the form:

$$\texttt{do}\, t_L = max(x_1, \ldots, x_p),\ min(y_1, \ldots, y_q),\ g$$

where the x_i's and y_i's are the loop bounds described above. If we can determine that the loop contains at most a single iteration, we perform the obvious simplifications to the code.

Within each interval i, we recursively generate code for level $L+1$. At level $L+1$ we only consider statements that were active at level L. This process continues until we reach level n, at which time we generate the elementary assignment statements.

5.3 Elementary statements

Once we have generated code for all levels, only a single statement will be active. We generate code to guard the statement from any conditions not already handled in loop bounds. If not all of the stride constraints could be expressed as loop steps, then these guards will contain mod expressions.

Finally, we output the transformed assignment statement. The statement has the same form as in the original code, except that the original index variables are replaced by expressions involving the new index variables. We determine these replacement expressions by using the Omega library to invert the mapping relation and extract expressions corresponding to each of the original index variables. For example, if the mapping is $[i_1, i_2] \rightarrow [i_1 + i_2, i_1]$ then i_1 is replaced by j_2 and i_2 is replaced by $j_1 - j_2$.

6 Related Work

The framework of Unimodular transformations [Ban90, WL91] has the same goal as our work, in that it attempts to provide a unified framework for describing loop transformations. It is limited by the facts that it can only be applied to perfectly nested loops, and that all statements in the loop nest are transformed in the same way. It therefore cannot represent some important transformations such as loop fusion, loop distribution and statement reordering. Unimodular transformations are generalized in [LP92, Ram92] to include mappings that are invertible but not unimodular. This allows the resulting programs to have steps in their loops, which can be useful for optimizing locality. Unimodular transformations are combined with blocking in [WL91, ST92]. A similar approach, although not using a unimodular framework, is described in [Wol89a].

Pugh [Pug91] gives techniques to represent loop fusion, loop distribution and statement reordering in addition to the transformations representable by unimodular transformations. Because it uses only single level affine schedules and requires that all dependences be carried by the outer loop, it can only be applied to programs that can be executed in linear time on a parallel machine. Methods to generate efficient code were not given.

Paul Feautrier [Fea92] generates schedules that are similar in form to our mappings but have slightly different semantics. His methods are designed to generate a schedule that produces code with a "maximal" amount of parallelism. These schedules will often not be optimal in practice because of issues such as granularity, data locality and code complexity. Our framework attempts to provide a setting in which multiple performance issues can be traded-off.

7 Implementation Status

The algorithms described in this paper are implemented in our extension of Michael Wolfe's `tiny` tool which is available via anonymous ftp from `ftp.cs.umd.edu` in the directory `pub/omega`.

8 Conclusions

We have presented a framework for unifying reordering transformations such as loop interchange, distribution, skewing, tiling, index set splitting and statement reordering. The framework is based on the idea that a transformation can be represented as a mapping from the original iteration space to a new iteration space. We have demonstrated that mappings are able to represent traditional reordering transformations, such as those above. We believe that using mappings is the purest or most fundamental way to describe arbitrary reordering transformations.

The framework is designed to provide a uniform way to represent and reason about transformations. The framework does not solve the fundamental problem of deciding which transformation to apply, but it does provide a simpler setting in which to solve this problem. We therefore believe that production systems would benefit from using our framework, rather than an arbitrary set of unrelated traditional transformations.

We have provided algorithms to test the legality of mappings, and to generate optimized code for mappings. Our code generation algorithm can be used to produce code that avoids and/or eliminates many of the guards that can occur around statements when performing reordering transformations. This makes our code generation algorithm useful for other applications such as the generation of code for distributed memory machines and the generation of code for traditional transformations.

References

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[Ban79] U. Banerjee. *Speedup of Ordinary Programs.* PhD thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, October 1979.

[Ban90] U. Banerjee. Unimodular transformations of double loops. In *Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 192–219, Irvine, CA, August 1990.

[CK92] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. In *Proceedings Supercomputing'92*, pages 114–125, Minneapolis, Minnesota, Nov 1992.

[Fea92] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II, Multidimensional time. *Int. J. of Parallel Programming*, 21(6), Dec 1992.

[KP94a] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. *Parallel Processing Letters*, 4(3):205–219, September 1994.

[KP94b] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *Lecture Notes in Computer Science 892: Seventh International Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994. Springer-Verlag.

[LP92] 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, Yale University, August 1992.

[Lu91] Lee-Chung Lu. A unified framework for systematic loop transformations. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 28–38, April 1991.

[Pug91] William Pugh. Uniform techniques for loop optimization. In *1991 International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[PW92] William Pugh and David Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-3191, Dept. of Computer Science, University of Maryland, College Park, December 1992.

[Ram92] J. Ramanujam. Non-unimodular transformations of nested loops. In *Supercomputing '92*, pages 214–223, November 1992.

[ST92] Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 175–187, San Francisco, California, Jun 1992.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.

[Wol89a] Michael Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, pages 655–664, November 1989.

[Wol89b] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London, 1989.

[Wol90] Michael Wolfe. Massive parallelism through program restructuring. In *Symposium on Frontiers on Massively Parallel Computation*, pages 407–415, October 1990.

[Wol91] Michael Wolfe. The tiny loop restructuring research tool. In *Proc of 1991 International Conference on Parallel Processing*, pages II–46 – II–53, 1991.