UMIACS-TR-93-134 November, 1992
CS-TR-3193 Revised April, 1993

A Framework for Unifying Reordering Transformations

Wayne Kelly
wak@cs.umd.edu
Dept. of Computer Science

William Pugh
pugh@cs.umd.edu
Institute for Advanced Computer Studies
Dept. of Computer Science

Univ. of Maryland, College Park, MD 20742

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 schedule that maps the original iteration space to a new iteration space. The framework is designed to provide a uniform way to represent and reason about transformations. As part of the framework, we provide algorithms to assist in the building and use of schedules. In particular, we provide algorithms to test the legality of schedules, to align schedules and to generate optimized code for schedules.

This work is supported by an NSF PYI grant CCR-9157384 and by a Packard Fellowship.

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, Pol88, 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, WL91a] 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:

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

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 can therefore not represent some important transformations such as loop fusion, loop distribution and statement reordering.

1.1 Schedules

The points in the iteration space resulting from a unimodular transformation will be executed in lexicographic order. Thus a unimodular transformation implicitly specifies a new order or *schedule* for the points in the original iteration space. We use this idea of a schedule as the basis for our unified reordering transformation framework. This framework is more general than unimodular transformations as it can describe a larger class of mappings (or schedules) from the old iteration space to the new iteration space.

A schedule has the following general form:

$$T : [i^1, \ldots, i^m] \rightarrow [f^1, \ldots, f^n] \mid C$$

where:

- The iteration variables $i^1, \ldots, i^m$ represent the loops nested around the statement(s).
- The $f^{j\prime}s$ are functions of the iteration variables.
- C is an optional restriction on the domain of the schedule.

This schedule 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 if condition C is true.
For example the above unimodular transformation would be represented by the schedule:

$$T : [i, j] \rightarrow [j, i]$$

In the case of unimodular transformations:

- All statements are mapped using the same schedule.
- The $f^{j\prime}s$ are linear functions of the iteration variables.

- The schedule is invertable and unimodular (i.e., 1-1 and onto).
- The dimensionality of the old and new iteration spaces are the same (i.e., $m = n$).
- There is no restriction C on the domain.

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

- We specify a separate schedule for each statement
- We allow the $f^{j'}s$ to include a constant term (possibly symbolic).
- We require the schedules to be invertable, but not necessarily unimodular (i.e., 1-1 but not necessarily onto).
- We allow the dimensionality of the old and new iteration spaces to be different.
- We allow the schedules to be piecewise (as suggested by [Lu91]): we can specify a schedule T_p as $\bigcup_i T_{pi} \mid C_{pi}$ where the $T_{pi} \mid C_{pi}$'s are schedules with disjoint domains.
- We allow the $f^{j'}s$ to be functions that include integer division and modular operations provided the denominator is a known integer constant.

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 alignment [ACK87]
- loop interleaving [ST92]
- loop blocking[1] (or tiling) [AK87]
- index set splitting[1] [Ban79]
- loop coalescing[1] [Pol88]
- loop scaling[1] [LP92]

1.2 Examples

Figure 1 gives some interesting examples of schedules.

1.3 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.

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

Code adapted from OLDA in Perfect club (TI) [B+89]

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)
```

Schedule (for parallelism)

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

Transformed code

```
     parallel do 20 mi = 1,morb
       parallel do 20 t2 = 1,np
         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)
```

Schedule (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 5 i = max(iB, k+1), min(iB+63, n)
5          a(i,kB)=a(i,kB)/a(kB,kB)
         do 20 t3 = kB+1, min(iB+62,n)
           do 10 k = kB, min(t3-1, kB+63)
             do 10 i = max(k+1,iB), min(iB+63,n)
10             a(i,t3)=a(i,t3)-a(i,k)*a(k,t3)
           do 20 i = max(iB,t3+1), min(iB+63,n)
20           if (t3<=kB+63) a(i,t3)=a(i,t3)/a(t3,t3)
         do 30 t3 = iB+63, n
           do 30 k = kB to min(iB+62,kB+63)
             do 30 i = max(k+1,iB), iB+63
30             a(i,t3)=a(i,t3)-a(i,k)*a(k,t3)
```

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)
```

Schedule (for parallelism)

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

Transformed code

```
     parallel do 10 i = 2,n
10     sum(i) = 0.
     do 30 t2 = 1, n-1
       parallel do 20 i = t2+1,n
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 [LP92] adapted from BLAS [JDH90]

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)
```

Schedule (for locality and parallelism)

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

Transformed code

```
      parallel do 10 t1 = 1, min(n,2*b-1)
        do 10 t2 = max(1-b,1-n), min(b-t1, n-t1)
          parallel do 10 k = max(1,t1+t2), min(n+t2,n)
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, Schedules, and Resulting Transformations

operation	Description	Definition
$F \circ G$	The composition of F with G	$x \rightarrow z \in F \circ G \Leftrightarrow \exists y$ s.t. $x \rightarrow y \in F \wedge y \rightarrow z \in G$
$F(S)$	Apply the relation F to the set S	$y \in F(S) \Leftrightarrow \exists x$ s.t. $x \rightarrow y \in F \wedge x \in S$
F^{-1}	The inverse of F	$x \rightarrow y \in F^{-1} \Leftrightarrow y \rightarrow x \in F$
$F \cap G$	The intersection of F and G	$x \rightarrow y \in F \cap G \Leftrightarrow x \rightarrow y \in F \wedge x \rightarrow y \in G$
$S \cap T$	The intersection of S and T	$x \in S \cap T \Leftrightarrow x \in S \wedge x \in T$
$domain(F)$	The domain of F	$x \in domain(F) \Leftrightarrow \exists y$ s.t. $x \rightarrow y \in F$
$range(F)$	The range of F	$y \in range(F) \Leftrightarrow \exists x$ s.t. $x \rightarrow y \in F$
$\pi_{1,\ldots,v}\ S$	The projection of S onto variables $1, \ldots, v$	$x \in \pi_{1,\ldots,v}\ S \Leftrightarrow \|x\| = v \wedge \exists y$ s.t. $xy \in S$
$feasible(S)$	True if S is not empty	$feasible(S) \Leftrightarrow \exists x \in S$

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

In Section 2 we describe how dependences and schedules are represented. In Section 3 we demonstrate that a large class of traditional transformations can be represented using schedules. In Section 4 we describe an algorithm that tests whether a schedule is legal. A surrounding system that is able to come up with a complete schedule by itself need only use this schedule legality test. Other systems may need the schedule component legality test and schedule alignment algorithm described in Section 5 to build a complete schedule.

In Section 6 we describe our code generation algorithm. This algorithm takes a schedule and produces optimized code corresponding to the transformation represented by that schedule. By making use of the gist operation [PW92] we are able to produce code with a minimal number of conditionals and loop bounds.

In Section 7 we extend our schedule syntax to allow us to denote the fact that a schedule produces fully-permutable loop nests [WL91a, WL91b]. Given a permutable schedule, it is easy to reorder or tile the loops for parallelism and locality without continual concern about legality.

In Section 8 we discuss surrounding systems and the interface between surrounding systems and our framework. Finally we discuss related work, give our implementation status and state our conclusions.

2 Representing Dependences and Schedules

Most of the previous work on program transformations uses data dependence directions and distances to summarize dependences between array references. For our purposes, these abstractions are too crude. We evaluate and represent dependences exactly using linear constraints over integer variables. We use the Omega test [Pug92, PW92] to manipulate and simplify these constraints. This approach allows us to accurately compose dependences and provides more information about which transformations are allowable.

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. Table 1 gives a brief description of the operations on integer tuple sets and relations that we have implemented.

See [Pug91] for a more thorough description.

2.2 Simple relations

Internally, 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 linear constraints. We can represent simple relations containing non-convex constraints such as $\{ [i] \to [i] \mid i \text{ even } \}$ by introducing wildcard variables (denoted by greek letters): $\{ [i] \to [i] \mid \exists \alpha \text{ s.t. } i = 2\alpha \}$.

2.3 Control dependence

We require that conditionals be removed using if-conversion [AKPW83] and 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 linear inequalities on the loop variables and symbolic constants.

Alternatively, structured if statements can be handled by treating them as atomic states.

In future research we plan handle a larger class of control dependences.

2.4 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.

2.5 Transitive dependence

If there is a dependence from $s_p[i]$ to $s_q[j]$ and from $s_q[j]$ to $s_r[k]$, then we say there is a transitive dependence from $s_p[i]$ to $s_r[k]$.

We calculate transitive dependences and store them in a graph called the transitive dependence graph. Our algorithms would work if applied to the normal dependence graph rather than the transitive dependence graph, but by using the transitive dependence graph we can determine earlier that a schedule is illegal.

Computing the transitive closure of the dependences can be expensive, and it is not always possible to find a closed form. Since the transitive dependences are needed only to improve efficiency, we can avoid computing a complete transitive closure when it appears too expensive or we can't find a closed form. A rough approximation of transitive closure works well for the applications to which we put it (the approximation has to be a lower bound on the transitive closure).

2.6 The gist and approx operations

We make use of the gist operation that was originally developed in [PW92]. 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 conjunction containing a minimal subset of the constraints in p such that ((gist p 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 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.

2.7 Schedules

We associate a separate schedule with each statement, we therefore need a way to refer to the schedules of individual statements. We represent the schedule 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 *schedule components*. For simplicity but without loss of generality we require that all of the schedules have n components. We refer to each of the positions $1, \ldots, n$ as *levels*. A level can also be thought of as all of the schedule components at a particular position.

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

3 Representing Traditional Transformations as Schedules

In this section we demonstrate how schedules 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 schedules to represent traditional transformations by describing how to modify schedules that correspond to the normal sequential execution of programs. When constructing these schedules we categorize schedule components as being either syntactic components (always an integer constant) or loop components (a function of the loop variables for 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 analagously).

The schedule that corresponds to the normal sequential execution 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 schedule($S, [] \rightarrow []$) returns a schedule for each of the assignment statements in S:

The common syntactic level (csl) of two statements s_p and s_q is defined as:

$$\text{csl}(s_p, s_q) \equiv \min\{i - 1 \mid 1 \leq i \wedge f_p^i \neq f_q^i \wedge \text{syntactic}(f_p^i) \wedge \text{syntactic}(f_q^i)\}$$

Intuitively, the common syntactic level of two statements is the deepest loop which surrounds both statements. Figure 2 describes how to construct schedules to represent traditional transformations by describing how to modify the schedules we have just described. Since the rules described in Figure 2 can be applied repeatedly, we can represent not only standard transformations but also any sequence of standard transformations.

4 Schedule Legality Test

In this section we describe an algorithm that tests whether a schedule is legal. A schedule 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.

The legality requirement is as follows: 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 $T_q(j)$:

$$\forall i, j, p, q, Sym\ i \rightarrow j \in d_{pq} \Rightarrow T_p(i) \prec T_q(j) \tag{1}$$

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

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

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

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

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

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

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

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

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

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, WL91b].

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

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: $\forall s_p, s_q \quad s_p \in D \wedge s_q \in D \Rightarrow \text{loop}(f_p^L) \wedge L \leq csl(s_p, s_q)) \wedge B$ 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 2: Using schedule transformations to achieve standard reordering transformations

where Sym is the set of all symbolic constants in the equation, and $\prec$ is the lexicographic ordering operator. We verify this by equivalently computing:

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

We also require that the schedule be 1-1 so that the new program performs exactly the same set of computations as the original program:

$$\forall p, q, i, j, Sym \; (p = q \wedge i = j) \Leftrightarrow T_p(i) = T_q(j) \tag{2}$$

which can be easily verified.

5 Aids to Building Schedules

A surrounding system that is able to come up with a complete schedule by itself need only use the schedule legality test. Other systems may need help in building a complete schedule. In this section we describe

```
function schedule(S, [i^1, ..., i^k] → [f^1, ..., f^a])
    case S of
        "for i^{k+1} = ... do S_1":
            return Schedule(S_1, [i^1, ..., i^k, i^{k+1}] → [f^1, ..., f^a, i^{k+1}])
        "S_1; S_2; ...; S_m":
            return ⋃_{p=1}^{m} Schedule(S_p, [i^1, ..., i^k] → [f^1, ..., f^a, p])
        "assignment #p":
            return {T_p : [i^1, ..., i^k] → [f^1, ..., f^a]}
```

the schedule component legality test and schedule alignment algorithm, which aid the surrounding system in building a complete schedule.

5.1 Level by level philosophy

Our algorithms that aid in the construction of schedules assume that the surrounding system is using the following basic philosophy. Schedules are constructed in a step by step manner:

1. Initially it is only known which statements are being scheduled and what their iteration variables are. None of the f_p^i's have been specified; we may not even know the depth (n) of the new iteration space.
2. At each step some of the unspecified f_p^i's are specified.
3. At each step we require that it be possible to extend the current partially specified schedule into a complete legal schedule by specifying the remaining unspecified f_p^i's.

The third requirement places some restrictions on the order in which the schedule components can be specified. For example if f_p^n was specified before specifying $f_p^1, \ldots, f_p^{n-1}$ then there would be no easy way to determine whether this partially specified schedule could be extended to a complete legal schedule. Therefore, we require that the schedule components be specified level by level starting at the first level, (i.e., during step k, $f_1^k, \ldots, f_p^k$ are specified). This strategy, is in some ways a generalization of Allen and Kennedy's `codegen` algorithm [AK87].

5.2 Ensuring legality

If we use T_p^k to represent the schedule:

$$[i_p^1, \ldots, i_p^{m_p}] \rightarrow [f_p^1, \ldots, f_p^k]$$

then after each step k we require that:

$$\forall i, j, p, q, Sym\ i \rightarrow j \in d_{pq} \ \Rightarrow T_p^k(i) \preceq T_q^k(j) \tag{3}$$

We can simplify this condition by noting that at level k we can ignore dependences carried at levels less than k:

$$\forall r, i, j, p, q, Sym\ 1 \leq r \leq k \wedge i \rightarrow j \in d_{pq}^r \Rightarrow t_p^r(i) \leq t_q^r(j) \tag{4}$$

$$\begin{aligned} \text{where} \quad t_p^r &= [i_p^1, \ldots, i_p^{m_p}] \rightarrow [f_p^r] \\ d_{pq}^1 &= d_{pq} \\ d_{pq}^r &= d_{pq}^{r-1} \cap \ ((t_q^{r-1})^{-1} \circ t_p^{r-1}) \end{aligned}$$

(if $i \rightarrow j \in d_{pq}^k$ then $T_p^{k-1}(i) \not\prec T_q^{k-1}(j)$, i.e., d_{pq}^k is the set of dependences in d_{pq} that haven't been satisfied before step k). If the requirement described by Equation 4 has been maintained during steps 1 through $k-1$ then at step k we need only ensure that:

$$\forall i, j, p, q, Sym\ i \rightarrow j \in d_{pq}^k \Rightarrow t_p^k(i) \le t_q^k(j) \tag{5}$$

The step by step process of specifying additional levels must continue until the schedule is 1-1. The condition that the schedule is 1-1 (Equation 2) together with Equation 3 and the following property of dependence relations:

$$\neg \exists i, p, Sym \text{ s.t. } i \rightarrow i \in d_{pp}$$

imply the legality condition (Equation 1).

5.3 Variable and constant parts

We distinguish two parts of a schedule component: the *variable part* and the *constant part*. The variable part is the largest subexpression of the schedule component that is a linear function of the iteration variables. The rest of the expression is called the constant part. For example in the schedule

$$[i, j] \rightarrow [2i + j + n + 1, 0, j]$$

the level 1 schedule component has variable part $2i + j$ and constant part $n + 1$.

The variable parts of a schedule are of primary importance in determining the parallelism and data locality of the code resulting from a schedule. The constant parts of a schedule will often affect the schedule's legality, but will generally have minimal affect on the resulting parallelism and locality.

It is therefore reasonable to require that the surrounding system specify the variable parts of the schedule, and let algorithms provided with our framework select constant parts that make the schedule legal.

5.4 Component legality test

The following sections describe how step k of the schedule construction process would proceed when using our schedule alignment algorithm. The surrounding system specifies the variable parts of the level k schedule components (one for each statement). Before we try to align (i.e., find constant parts for) the variable parts, we can test each variable part to see if it is "legal" in isolation. A variable part V_p^k is legal for a statement s_p, if using it as a schedule component would not violate any self-dependences[1] on s_p, including any transitive self-dependences.

If a variable part is illegal for a given statement at a given level then it cannot be used in any schedule component in that context.

More formally, a variable part V_p^k for statement s_p is considered legal at level k iff:

$$\forall i, j, Sym\ i \rightarrow j \in d_{pp}^k \Rightarrow v_p^k(i) \le v_p^k(j) \tag{6}$$

where $v_p^k : [i_p^1, \ldots, i_p^{m_p}] \rightarrow [V_p^k]$ and d_{pq}^k is defined as it was in Equation 4.

For example, in the following code i and j are both legal variable parts for statement 1 at level 1, but $-i$ and $-j$ are illegal.

```
     do 2 i = 1, n
       do 2 j = 1, n
1        a(i,j) = b(i,j-1) + a(i-1,j)
2        b(i,j) = a(i,j-1)
```

[1] A self-dependence is a dependence from one iteration of a statement to another iteration of the same statement.

Using legal variable parts is necessary but not sufficient to ensure that they can be aligned. For example i is a legal variable part for statement 1 at level 1 and j is a legal variable part for statement 2 at level 1, but they cannot be aligned with one another.

If we are not able (or willing) to compute the exact transitive closure, then it is important that the approximation we use be a subset of the actual dependences. This approximation may cause some illegal variable parts to be accepted but importantly won't reject any legal variable parts. It is acceptable to accept illegal variable parts at this stage as we will latter determine that they are illegal. In other words, the component legality test is only an initial filter designed to improve the efficiency of the rest of the algorithm.

5.5 Schedule alignment algorithm

We now assume that we have been given a legal variable part V_p^k for each statement s_p. It is our job to, if possible, select constant parts that align the variable parts (i.e., satisfy Equation 5).

We create a new variable c_p^k for each statement s_p. These new variables represent the constant offsets that must be added to the variable parts to make them align with one another. More precisely, this can be stated as $f_p^k = V_p^k + c_p^k$. We construct a set of constraints involving these constant offset variables, such that any set of constant offset values that satisfy the constraints will properly align the variable parts.

We first consider the constraints on a pair of constant offset variables c_p^k and c_q^k, that are imposed by a simple dependence relation $d \subseteq d_{pq}^k$. Equation 5 tells us that:

$$\forall i, j, Sym\ i \rightarrow j \in d \Rightarrow t_p^k(i) \leq t_q^k(j)$$

By substituting $v_p^k(i) + c_p^k$ for t_p^k and removing the quantification on Sym, we get:

$$\forall i, j \text{ s.t. } i \rightarrow j \in d \Rightarrow v_p^k(i) + c_p^k \leq v_q^k(j) + c_q^k$$

which is the set of constraints on c_p^k, c_q^k and the symbolic constants that are imposed by the simple dependence relation d. These constraints can be described equivalently as:

$$A: \quad \neg(\exists i, j \text{ s.t. } i \rightarrow j \in d \wedge v_p^k(i) + c_p^k > v_q^k(j) + c_q^k) \tag{7}$$

Unfortunately, the negation in Equation 7 usually produces a disjunction of several constraints. Our goal now, is to deduce from 7 a system of linear constraints for the alignment constants. The conditions, D, under which the dependence exists are:

$$D: \quad \exists i, j \text{ s.t. } i \rightarrow j \in d \tag{8}$$

Since $\neg D \Rightarrow A$, we know that $A \equiv (D \Rightarrow A)$. We transform A as follows:

$$\begin{aligned} A &\equiv D \Rightarrow A \\ &\equiv \neg(D \wedge \neg A) \\ &\equiv \neg(D \wedge \text{gist } \neg A \text{ given } D) \\ &\equiv D \Rightarrow \neg\text{gist } \neg A \text{ given } D \end{aligned}$$

Therefore Equation 7 is equivalent to:

$$\exists i, j \text{ s.t. } i \rightarrow j \in d \Rightarrow \neg(\text{gist } \exists i, j \text{ s.t. } i \rightarrow j \in d \wedge v_p^k(i) + c_p^k > v_q^k(j) + c_q^k \text{ given } \exists i, j \text{ s.t. } i \rightarrow j \in d) \tag{9}$$

Usually, the gist in Equation 9 will produce a single inequality constraint. There are cases where this does not occur. For example, in the following code,

```
    do 2 i = 1, min(n,m)
1     a(i) = ...
2     ...  = ... a(i) ...
```

attempting to align $V_1^1 = i$ and $V_2^1 = 0$ gives:

$$1 \leq n \wedge 1 \leq m \Rightarrow (c_2^1 - c_1^1 \geq \mathtt{n} \vee c_2^1 - c_1^1 \geq \mathtt{m})$$

In practice we have found that this very seldomly occurs. When the gist produces a disjunction of inequality constraints, we strengthen the condition by throwing away all but one of the inequalities produced by the gist.

Unfortunately, Equation 9 also contains an implication operator. We describe two approaches to finding constant terms to satisfy Equation 9. The first, described in Section 5.5.1, is a fast and simple method that seems to handle the problems that arise in practice. The second, described in Section 5.5.2, is a more sophisticated method that is complete provided the gist in Equation 9 produces a single inequality constraint.

5.5.1 A fast and simple but incomplete technique

Rather than constructing the set of constraints described by Equation 9, we construct a slightly stronger set of constraints by changing the antecedent to true:

$$\neg(\text{gist } \exists i, j \text{ s.t. } i \rightarrow j \in d \wedge v_p^k(i) + c_p^k > v_q^k(j) + c_q^k \text{ given } \exists i, j \text{ s.t. } i \rightarrow j \in d) \tag{10}$$

If an acceptable set of constant offset values can be found that satisfy these stronger constraints, these offsets must satisfy the weaker constraints, and therefore align the variable parts.

For a given pair of statements s_p and s_q, we form a single set of constraints A_{pq}^k by combining the alignment constraints (Equation 10) resulting from all simple dependence relations between those two statements. We then combine the A_{pq}^k constraints one statement at a time, checking at each stage that the alignment constraints formed so far are satisfiable for all values of the symbolic constants:

function AlignSchedule (vp_being_considered)
  for each statement p
    v_p^k = vp_being_considered[p]
  Conditions = true
  for each statement p
    for each statement q $< p$
      Calculate A_{pq}^k
      Conditions = Conditions $\wedge A_{pq}^k$
      if not ($\forall Sym$ ($\exists c_1^k, \ldots, c_p^k$ s.t. Conditions) then
        return false
  return Conditions

Figure 3 gives an example of aligning schedules using this technique.

Having obtained a set of alignment constraints, we can either return this set of constraints for use by an external system, or we can find a set of constant offset values that satisfy the alignment constraints. In finding this set of satisfying values, we could consider optimality criteria such as locality or lack of loop carried dependences.

Original Code:

```
      do 2 i = 1, n
        do 2 j = 1, n
1         a(i,j) = b(i,j-2)
2         b(i,j) = a(i,j-3)
```

Example of variable parts being considered at level one: $v_1^1[i,j] = j, \quad v_2^1[i,j] = j$

(We use the normal dependence graph rather than the transitive dependence graph)

Alignment constraints from s_1 to s_2:

$$d_{12}^1 \quad : \{[i,j] \rightarrow [i,j+3] \mid \; 1 \leq i \leq n; 1 \leq j \leq n-3\}$$

$$\begin{aligned}
A_{12}^1 \;\Leftrightarrow\;& \neg\,(\text{gist } \exists [i,j],[i',j'] \text{ s.t. } [i,j] \rightarrow [i',j'] \in d \wedge v_1^1([i,j]) + c_1^1 > v_2^1([i',j']) + c_2^1 \\
& \quad \text{given } \exists [i,j],[i',j'] \text{ s.t. } [i,j] \rightarrow [i',j'] \in d) \\
\Leftrightarrow\;& \neg\,(\text{gist } \exists [i,j],[i',j'] \text{ s.t. } i' = i \wedge j' = j+3 \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n-3 \wedge j + c_1^1 > j' + c_2^1 \\
& \quad \text{given } \exists [i,j],[i',j'] \text{ s.t. } i' = i \wedge j' = j+3 \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n-3) \\
\Leftrightarrow\;& \neg\,(\text{gist } 4 \leq n \wedge 4 + c_2^1 \leq c_1^1 \text{ given } 4 \leq n) \quad \text{/* simplified using the Omega test */} \\
\Leftrightarrow\;& \neg\,(4 + c_2^1 \leq c_1^1) \\
\Leftrightarrow\;& c_1^1 \leq 3 + c_2^1
\end{aligned}$$

Alignment constraints from s_2 to s_1:

$$d_{21}^1 \quad : \{[i,j] \rightarrow [i,j+2] \mid \; 1 \leq i \leq n; 1 \leq j \leq n-2\}$$

$$\begin{aligned}
A_{21}^1 \;\Leftrightarrow\;& \neg\,(\text{gist } \exists [i,j],[i',j'] \text{ s.t. } [i,j] \rightarrow [i',j'] \in d \wedge v_2^1([i,j]) + c_2^1 > v_1^1([i',j']) + c_1^1 \\
& \quad \text{given } \exists [i,j],[i',j'] \text{ s.t. } [i,j] \rightarrow [i',j'] \in d) \\
\Leftrightarrow\;& \neg\,(\text{gist } \exists [i,j],[i',j'] \text{ s.t. } i' = i \wedge j' = j+2 \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n-2 \wedge j + c_2^1 > j' + c_1^1 \\
& \quad \text{given } \exists [i,j][i',j'] \text{ s.t. } i' = i \wedge j' = j+2 \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n-2) \\
\Leftrightarrow\;& \neg\,(\text{gist } 3 \leq n \wedge 3 + c_1^1 \leq c_2^1 \text{ given } 3 \leq n) \quad \text{/* simplified using the Omega test */} \\
\Leftrightarrow\;& \neg\,(3 + c_1^1 \leq c_2^1) \\
\Leftrightarrow\;& c_2^1 \leq 2 + c_1^1
\end{aligned}$$

These variable parts can be aligned because:

$$\begin{aligned}
& \forall Sym\ (\exists c_1^1, c_2^1 \text{ s.t. } A_{12}^1 \wedge A_{21}^1) \\
\Leftrightarrow\;& \forall n\ (\exists c_2^1, c_1^1 \text{ s.t. } c_1^1 \leq 3 + c_2^1 \wedge c_2^1 \leq 2 + c_1^1) \\
\Leftrightarrow\;& \forall n\ (\exists c_2^1, c_1^1 \text{ s.t. } c_2^1 - 2 \leq c_1^1 \leq c_2^1 + 3) \\
\Leftrightarrow\;& \forall n\ True \\
\Leftrightarrow\;& True
\end{aligned}$$

Figure 3: Example of Aligning Schedules

5.5.2 A complete technique

If we wish to construct exactly the set of constraints described by Equation 9 then we can use the following techniques proposed by Quinton [Qui87]. The vertex method [Qui87] relies on the fact that a linear constraint holds everywhere inside a polyhedron if and only if it holds at all vertices of the polyhedron.

Therefore, we can convert Equation 9 into a conjunction of constraints by determining the vertices of Equation 8 and adding the constraint that the consequent of Equation 9 is true at each of those points. To obtain constraints on the c_p^k's in terms of the other symbolic constants, we represent each c_p^k as: $\sum_{i=1}^{m} \lambda_{pi} S_i + \lambda_{p0}$ where the S_i's are the original symbolic variables and the λ_{pi}'s are new variables. Constraints are then formed on the λ_{pi}'s rather than the c_p^k's. We combine the constraints generated from all dependence relations, and then use any integer programming technique to find a solution for the λ_{pi}'s. This solution for the λ_{pi}'s can then be used to form a solution for the c_p^k's.

6 Optimized Code Generation

In this section we describe an algorithm to generate efficient source code for a schedule. As an example, we consider changing the KIJ version of Gaussian Elimination (without pivoting) into the IJK version.

```
      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)
```

The version refers to the nesting of the loops around the inner most statement. Michael Wolfe notes that this transformation requires imperfect triangular loop interchange, distribution, and index set splitting [Wol91]. We can also produce the IJK ordering using the following schedule:

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

A naive code generation strategy that only examined the minimum and maximum value at each level 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. 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 schedules in this section (they can be handled by considering each piece of the schedule as a separate statement).

6.1 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 schedule associated with statement s_p is T_p, then the statement's new iteration space J_p is given by $T_p(I_p)$. This new iteration space is represented internally as a set of linear constraints and is the starting point for the rest of this section.

In the example above, the original iteration space is:

$$\begin{array}{ll} I_{10}: & \{[k,i] | 1 \leq k \leq n \wedge k+1 \leq i \leq n\} \\ I_{20}: & \{[k,i,j] | 1 \leq k \leq n \wedge k+1 \leq i \leq n \wedge k+1 \leq j \leq n\} \end{array}$$

and the new iteration space is:

$$\begin{array}{ll} I_{10}: & \{[i,k,1,0] | 1 \leq k \leq n \wedge k+1 \leq i \leq n\} \\ I_{20}: & \{[i,j,0,k] | 1 \leq k \leq n \wedge k+1 \leq i \leq n \wedge k+1 \leq j \leq n\} \end{array}$$

6.2 Code generation for a single level

Our code generation algorithm builds the transformed code recursively, level by 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 j^L to be used for this level. For each statement s_p, we need to generate a set of constraints J_p^L that represents the values of j^L for which statement s_p should be executed. We cannot simply project J_p onto j^L, because this will only give us absolute upper and lower bounds on j^L. Expressing the tightest possible upper and lower bounds on j^L may require using expressions that involve index variables from earlier levels. So we instead calculate the projection:

$$\pi_{1,\ldots,L}(J_p)$$

This set contains constraints on j^L, on index variables from earlier levels $(j^1, \ldots, j^{L-1})$, 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 know about the values of index variables from earlier levels. So we have

$$J_p^L \;=\; \text{gist } \; \pi_{1,\ldots,L}(J_p) \; \text{ given } \; known^{L-1}$$

The J_p^L sets for different statements may, in general, overlap. If J_p^L and J_q^L overlap over some range, then the j^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^L sets of the statements in the loop. The problem is that in general the J_p^L sets will have incompatible constraints. Modulo constraints are constraints of the form $j^L = c\alpha + s$, where α is a wildcard variable. These sorts of constraints can appear in J_p^L if the coefficients of the schedule components are not ± 1, or if the original program contains steps which are not ± 1.

```
procedure GenerateCode()
    for each (stmt)
        J[stmt] = T[stmt](I[stmt])
    GenerateCodeRecursively(1, {all stmts}, True, True)

procedure GenerateCodeRecursively(level, was_active, required, known)
    for each (stmt) ∈ was_active
        JL[stmt] = gist  π_{1,...,level} J[stmt]  given  known
        M[stmt] = approx  (JL[stmt,level])
    for each (interval) in temporal order
        R = required ∧(⋀_{stmt∈was_active} appropriate_range(stmt, interval) ∧ greatest_common_step(interval,
  JL)
        if | {stmt : stmt is active in interval} | = 1
            R = R ∧ J[stmt active in interval]
        R = gist  R  given  known
        if feasible(R)
            for each (stmt)
                active[stmt] = was_active[stmt] ∧ (stmt is active in interval) ∧ feasible(JL[stmt] ∧ R ∧ known)
            parallel = no dependences between active statements carried at this level
            output_loop(level, R, parallel)
            new_known = known ∧ information in R represented by loop
            new_required = gist  R  given  new_known
            if last level
                stmt = only active statement
                output_guard(new_required)
                output_statement(stmt)
            else
                GenerateCodeRecursively(level+1, active, new_required, new_known)
```

Figure 4: The Code Generation algorithm

We solve this problem by removing all modulo constraints from the J_p^L sets, and to worry about adding them later. To remove the modulo constraints, we use approx :

$$M_p^L = \text{approx}\,(J_p^L)$$

The constraints in J_p^L now describe a continuous range of values for j^L. For purposes of explanation we define (but do not compute):

$$E^L = \bigcup_p M_p^L$$

Having removed the modulo 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 j^L loop iterating over all points in E and containing all statements, then we would possibly still execute some statements with values of j^L not in their M_p^L ranges. We could overcome this problem by adding guards around the elementary assignment statements. However, we prefer a more efficient solution.

We would like to partition E^L into disjoint intervals such that, if a statement is active at any point in an interval, then it is active at every point in that interval. We could then generate a separate j^L loop for each interval, and put into those loops exactly the set of statements that are active at every point in that loop. If we did so we would not need to add guards around elementary assignment statements, because the constraints specified in the M_p^L ranges would be represented entirely by the bounds of the loops.

Unfortunately, it is not always possible to partition E^L in this way, because we may not know at compile time how the execution periods of statements relate to one another. This is the case when the bounds in the M_p^L ranges involve symbolic constants or `min`s and `max`s. In these cases we represent as much information as possible in the loop bounds, and represent any remaining information in guards around the elementary assignment statements.

In general each of the M_p^L ranges will have multiple (non-redundant) upper and lower bounds. For each statement s_p we arbitrarily choose a lower bound l_p^L and an upper bound u_p^L from M_p^L. These are the bounds that will be represented in the loop bounds. The remaining bounds if any will be represented in guards. For each statement s_p, the two points l_p^L and u_p^L divide E^L into three disjoint intervals: B_p^L, D_p^L and A_p^L, that are "before", "during" and "after" the execution of the statement respectively.

$$\begin{array}{rl} B_p^L: & \{t \mid t < l_p^L\} \\ D_p^L: & \{t \mid l_p^L \le t \ \wedge t \le u_p^L\} \\ A_p^L: & \{t \mid l_p^L \le t \ \wedge u_p^L < t\} \end{array}$$

We partition E into disjoint intervals by forming combinations of these intervals from different statements. Each combination is made up of one of B_p^L, D_p^L or A_p^L from each statement s_p. For example if we had two statements we would enumerate the following intervals:

$$\begin{array}{ccccc} B_1^L B_2^L & \rightarrow & B_1^L D_2^L & \rightarrow & B_1^L A_2^L \\ \downarrow & & \downarrow & & \downarrow \\ D_1^L B_2^L & \rightarrow & D_1^L D_2^L & \rightarrow & D_1^L A_2^L \\ \downarrow & & \downarrow & & \downarrow \\ A_1^L B_2^L & \rightarrow & A_1^L D_2^L & \rightarrow & A_1^L A_2^L \end{array}$$

A statement s_p is said to be active in an interval if that interval was formed using D_p^L. Intervals in which there are no active statements (e.g., $B_1^L B_2^L$ or $B_1^L A_2^L$) are not actually enumerated as code does not have to be generated for these intervals. There exists an obvious partial order on intervals (as shown by the arrows above), and we generate the code for the intervals in a total order compatible with that partial order.

For each interval i we calculate C_i^L — the range of values of j^L corresponding to the interval. C_i^L is determined by intersecting the appropriate intervals of the statements. The appropriate interval for s_p is either B_p^L, D_p^L or A_p^L, depending on whether s_p is before, during or after in this interval.

Now that we know exactly which statements are active in a particular interval, we may be able to add some of the modulo constraints that we removed earlier. Each statement s_p that is active in this interval may contribute a single modulo constraint that was removed earlier:

$$j^L = a_p^L \beta_p^L + b_p^L$$

where a_p^L is a constant, β_p^L is a wildcard variable and b_p^L is a constant term (possibly involving symbolic constants).

The *greatest common step* of this interval is:

$$gcs_i^L = gcd(\{a_p^L \mid s_p \text{ is active}\} \cup \{gcd(b_q^L - b_p) \mid s_q \text{ is active} \wedge s_p \text{ is active}\})$$

In performing these calculations, the gcd of an expression is defined to be the gcd of all of the coefficients in the expression.

We pick an arbitrary statement s_p that is active in the interval, and add to C_i^L the modulo constraint:

$$j^L = gcs_i^L \ \beta + b_p^L$$

We can safely add this constraint since if j^L satisfies the modulo constraint of any active statement then it will also satisfy this constraint.

The greatest common step will later be extracted from these constraints and used as the step for the loop corresponding to this interval. In general the step will not enforce all of the modulo constraints, but it is the best we can do at this level. The remaining modulo constraints (if any) will have to be enforced at deeper levels.

Before actually generating code for this interval, we must check that the proposed range C_i^L is consistent with the information that is known about the index variables at earlier levels. If $(C_i^L \ \wedge \ known^{L-1})$ is not feasible then code is not generated for this interval. If $(C_i^L \ \wedge \ known^{L-1})$ is feasible then we may be able to simplify the constraints in C_i^L by making use of the information in $known^{L-1}$. We calculate

$$R_i^L = \text{gist } C_i^L \text{ given } known^{L-1}$$

If R_i^L is feasible we generate a `do` loop to iterate over the appropriate values of j^L. If we earlier added a modulo constraint to C_i^L then we may still have a modulo constraint of the form:

$$j^L = g \ \beta + c$$

. If this is the case we enforce the 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 modulo constraint.

The loop's lower bound is derived from constraints in R_i^L of the form: $lower \leq m \ j^L$. If the following conditions are true:

$$\begin{aligned} known^{L-1} &\Rightarrow \quad lower = m\beta \\ lower = m \ j^L \wedge known^{L-1} &\Rightarrow \quad j^L = \gamma \ g + c \end{aligned}$$

then we can use:

$$lower/m$$

as a lower bound of the loop, otherwise we will have to use:

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

(if R_i^L doesn't contain a modulo constraint then g is 1 and c is 0)

The loop's upper bound is derived from constraints in R_i^L of the form: $n\ j^L \leq upper$. 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:

$$\begin{array}{ll} \text{do} \quad j^L = & max(x_1, \ldots, x_p),\ min(y_1, \ldots, y_q),\ g \\ & \ldots \end{array}$$

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.

We generate a sequential loop if there exists a data dependence that is carried by the loop. Otherwise we generate a parallel loop.

Within each interval i which contains at least one iteration, 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.

6.3 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 modulo 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 test to invert the scheduling relation and extract expressions corresponding to each of the original index variables. For example, if the schedule 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 $j_1 - j_2$.

7 Permutable Schedules

Some codes have only a handful of legal schedules, while other codes have an enormous number of legal schedules. LU decomposition is a typical example of code with a large number of legal schedules.

```
      do 20 k = 1, n
        do 20 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)
```

A large number of legal schedules exist, including:

$$\begin{array}{l}
T_{10} : [k, i] \rightarrow [k, i, k],\ \ T_{20} : [k, i, j] \rightarrow [k, i, j] \\
T_{10} : [k, i] \rightarrow [k, k, i],\ \ T_{20} : [k, i, j] \rightarrow [k, j, i] \\
T_{10} : [k, i] \rightarrow [i, k, k],\ \ T_{20} : [k, i, j] \rightarrow [i, k, j] \\
T_{10} : [k, i] \rightarrow [i, k, k],\ \ T_{20} : [k, i, j] \rightarrow [i, j, k] \\
T_{10} : [k, i] \rightarrow [k, k, i],\ \ T_{20} : [k, i, j] \rightarrow [j, k, i] \\
T_{10} : [k, i] \rightarrow [k, i, k],\ \ T_{20} : [k, i, j] \rightarrow [j, i, k]
\end{array}$$

This situation is common and is typified by a nested set of adjacent loops that are fully permutable [WL91a, WL91b]. A set of loops is *fully permutable* iff all permutations of the loops are legal. We have developed an extension to our schedule syntax that allows us to succinctly describe a large number of related schedules. Expressions in this extended syntax are called *permutable schedules.* A permutable schedule represents a set of normal schedules and has the following general form:

$$T_p : [i_p^1, \ldots, i_p^{m_p}] \rightarrow [g_p^1, \ldots, g_p^{n_p}]$$

where the g_p^j are either schedule components or permutations lists. *Permutation lists* have the form

$$\langle e_1, \ldots, e_v \rangle_x$$

where the e_j are schedule components. A permutable schedule represents all of the normal schedules that can be obtained by replacing each of its permutation lists by some permutation of the schedule components in that permutation list. For example the permutable schedule:

$$[k, i, j] \rightarrow [k, \langle i, j \rangle, 1]$$

represents the following two normal schedules:

$$[k, i, j] \rightarrow [k, j, i, 1] \text{ and } [k, i, j] \rightarrow [k, i, j, 1]$$

A permutable schedule is legal if and only if all of the schedules it represents are legal. This fact allows us to prove an important property of permutable schedules: all schedules represented by a legal permutable schedule will produce fully permutable loop nests. Recognizing that a schedule will produce fully permutable loop nests is useful for a number of reasons, including those described in Section 7.1.

It is not unusual to find multiple statements, each of which has a set of fully permutable loop nests, but due to alignment constraints choosing any permutation for one of the statement results in only one of the permutations for the other statements being legal. In this situation we still want to use our permutation syntax, but we need to indicate that the permutable schedule only represents those schedules that can be obtained by using the same permutation for all of these permutation lists. We indicate this by giving these permutation lists the same subscript. For example the following is a legal permutable schedule for LU decomposition:

$$T_{10} : [k, i] \rightarrow [\langle k, k, i \rangle_1], \quad T_{20} : [k, i, j] \rightarrow [\langle k, j, i \rangle_1]$$

We use the term *permutation list set* to refer to a set of permutation lists with the same subscript.

7.1 Schedules for blocking/tiling

Direct generation of blocked or tiled loops is only possible if there exists a fully permutable loop nest [WL91a, Wol89a]. A permutable schedule represents a set of schedules, all of which produce fully permutable loop nests. It is therefore easy to build a schedule corresponding to a blocking transformation from a permutable schedule.

Given a fully permutable loop nest, we need to decide which loops will be blocked, what their blocking factors will be, and which permutation of the loops will be used. These choices must be made consistently for all permutation lists in a permutation list set. We have therefore developed a syntax that specifies these blocking specifications for a permutation list set. For a permutation list set x, with v positions, a blocking specification has the form:

$$x : \ [h_1, \ldots, h_w]$$

The h_j expressions have either the unblocked form k or the blocked form $k\!:\!c$ where k is a position $(1, \ldots, v)$ and c is a blocking factor (a known integer constant[2]). The blocked expressions specify which loops will be blocked and what their blocking factors will be. The order of the expressions specifies which permutation

[2]We are currently working on techniques to allow symbolic constants to be used.

```
procedure BuildSchedule(Level)
    for each Statement
        LegalVariableParts = {list of profitable legal variable parts}
    for each Combination of LegalVariableParts (1 from each statement)
        AlignConstraints = AlignSchedule(Combination)
        AlignedSchedules = {set of aligned schedules derived from AlignConstraints}
        for each AlignedSchedule
            if AlignedSchedule is complete then
                if {WorthAccepting} then
                    add AlignedSchedule to list of accepted schedules
            else
                if {WorthContinuing} then
                    BuildSchedule(Level+1)

Start by calling: BuildSchedule(1)
```

Figure 5: General form of surrounding system, in collaborative setting

of the loops will be used. Every position must appear exactly once as an unblocked expression, and any blocked instance of a loop must come before the unblocked instance. For example the following is a blocking specification for the LU decomposition permutable schedule above:

$$1: \ [1\!:\!64,\ 3\!:\!64,\ 2,\ 1,\ 3]$$

Given a permutable schedule and a blocking specification we can build a schedule that produces blocked code as follows: We use the schedule components outside of the permutation lists unchanged. We replace each permutation list by a number of normal levels, creating a new level for each entry h_i in the blocking specification for that permutation list. For an unblocked expression k we use the k'th schedule component in the permutation list. For a blocked expression $k : c$ we use $c\ ((E - L)\ div\ c) + L$, where E is the k'th schedule component in the permutation list, and L is a constant expression chosen by our system to simplify the loop bounds. For example, the above blocking specification will produce the following schedule:

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

which produces the code given in Figure 1.

8 The Surrounding System

Our framework is designed to provide a uniform way to represent and reason about 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 parallelizing environment or an automatic parallelizing compiler. This surrounding system is finally responsible for deciding which transformation should be applied. In this section we discuss surrounding systems and the interface between surrounding systems and our framework.

Our framework can be used in two different settings. In the first of these settings, the surrounding system interacts with algorithms of Section 5 to build a schedule. In the second setting, the surrounding system decides on a transformation by itself, that is then represented as a schedule and used to generate code using our code generation algorithm.

8.1 Collaborative schedule generation

As mentioned in Section 5, it is reasonable to require that the surrounding system specify the variable parts of the schedule, and allow algorithms provided with the framework to select constant parts that make the schedule legal. Figure 5 gives the general form of a surrounding system in such a setting. The code fragments in curly braces are the parts that would change from one implementation to another. In its most general form, this is a recursive backtracking algorithm. It is therefore capable of generating more than one schedule. By generating a set of schedules rather than a single schedule we have a greater chance of finding the "best" schedule. Of course the problem then is to decide which of the generated schedules to use. If the generated set of schedules is relatively small, a schedule can be chosen by applying traditional performance estimation or by having the user select among a set of transformed codes.

If the code fragments in curly braces consider only one legal variable part per statement and only one alignment per set of alignment constraints, then no backtracking will occur and only one schedule will be generated. Such an implementation would be potentially very efficient. However, to find the "best" transformation, the algorithms that choose the legal variable parts and the alignments would have to be very intelligent.

8.2 Using permutable schedules

In circumstances where a large number of legal schedules exist, we first generate permutable schedules. These permutable schedules can then be used to generate code that is optimized for parallelism and locality. This approach has the advantage that fewer combinations are considered when generating permutable schedules. Using permutable schedules to generate optimized code is also easy because we know that all permutations are legal, so we can concentrate on performance issues while ignoring legality.

9 Related Work

The framework of Unimodular transformations [Ban90, WL91a, ST92, KKB92] 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 can therefore not 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 invertable 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 [WL91a, ST92]. A similar approach, although not using a unimodular framework, is described in [Wol89a].

Lu describes in [Lu91] a classification of scheduling techniques into various generality classes. Using their classification scheme, our schedules fit into the Mixed-Nonuniform class which is the most general class.

Our previous paper [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. It uses less sophisticated methods for aligning schedules than our current techniques, and does not give methods to generate efficient code.

Paul Feautrier [Fea92a, Fea92b] generates the same type of schedules that we do (generating a separate schedule for each statement). His methods are designed to generate a single 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. Feautrier does not give methods for generating code corresponding to the schedules.

10 Implementation Status

Prototype versions of most of the algorithms described in this paper are currently implemented in our extension of Michael Wolfe's `tiny` tool, and we are continuing to expand and strengthen our implementation. Our extension of `tiny` is available via anonymous ftp from `ftp.cs.umd.edu` in the directory `pub/omega`.

11 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 schedule that maps the original iteration space to a new iteration space. We have demonstrated that schedules are able to represent traditional reordering transformations, such as those above. We believe that using schedules 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 that assist in the building and use of schedules. In particular we have provided algorithms to test the legality of schedules, to align schedules, and to generate optimized code for schedules. 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

[ACK87] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[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.

[AKPW83] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conf. Rec. Tenth ACM Symp. on Principles of Programming Languages*, pages 177–189, January 1983.

[B+89] M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, March 1989.

[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.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part I, One-dimensional time. *Int. J. of Parallel Programming*, 21(5), Oct 1992. Postscript available as `pub.ibp.fr:ibp/reports/masi.92/78.ps.Z`.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II, Multidimensional time. *Int. J. of Parallel Programming*, 21(6), Dec 1992. Postscript available as `pub.ibp.fr:ibp/reports/masi.92/28.ps.Z`.

[JDH90] I. Duff J.J. Dongarra, J. DuCroz and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. on Math. Soft.*, 16:1–17, March 1990.

[KKB92] K. G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hieracical parallel machines in polynomial time. In *Proc. of the 1992 International Conference on Supercomputing*, pages 82–92, July 1992.

[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.

[Pol88] C. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[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. An earlier version of this paper appeared at the SIGPLAN PLDI'92 conference.

[Qui87] Patrice Quinton. The systematic design of systolic arrays. In F. Fogelman, Y. Robert, and M. Tschuente, editors, *Automata networks in Computer Science*, pages 229–260. Manchester University Press, December 1987.

[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.

[WL91a] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.

[WL91b] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. In *IEEE Transactions on Parallel and Distributed Systems*, July 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.