Finding Legal Reordering Transformations using Mappings

Wayne Kelly William Pugh

wak@cs.umd.edu, (301)-405-2726 pugh@cs.umd.edu, (301)-405-2705

Dept. of Computer Science

Univ. of Maryland, College Park, MD 20742

April 29, 1994

Abstract

Traditionally, optimizing compilers attempt to improve the performance of programs by applying source to source transformations, such as loop interchange, loop skewing and loop distribution. Each of these transformations has its own special legality checks and transformation rules which make it hard to analyze or predict the effects of compositions of these transformations. To overcome these problems we have developed a framework for unifying iteration reordering transformations. The framework is based on the idea that all reordering 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. An optimizing compiler would use our framework by finding a mapping that both corresponds to a legal transformation and produces efficient code. We present the mapping selection problem as a search problem by decomposing it into a sequence of smaller choices. We then characterize the set of all legal mappings by defining an implicit search tree.

1 Introduction

Traditionally, optimizing compilers attempt to parallelize programs and improve their performance, by applying source to source transformations, such as loop interchange, loop skewing and loop distribution [Wol89a]. Each of these transformations has its own special legality checks and transformation rules. This makes it difficult to find sequences of transformations that obtain some desired goal.

To overcome these problems, many researchers have proposed frameworks that unify some of these reordering transformations [Ban90, WL91, LMQ91, LP92, Ram92, Fea92a, Fea92b, ST92, DR92]. We have developed a framework that handles more reordering transformations than most existing transformation frameworks. Our 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. Our framework is more expressive than most because we allow a separate mapping to be associated with each statement and do not require that the mappings be unimodular.

Like all other frameworks, we need to be able to perform the following three tasks:

- Determine which mappings correspond to legal transformations.
- Determine which of these legal mappings will produce efficient code.
- Generate the transformed source code from a given mapping.

Unfortunately, but not surprisingly, the extra expressiveness of our framework makes it much harder for us to perform these tasks compared to simpler frameworks.

We will present the combination of the first two of these tasks as a search problem. The first task corresponds to defining the search tree and the second task corresponds to selecting a path within that tree. This paper will deal almost entirely with the first task, i.e. defining the search tree. Interested readers are referred to our previous work [KP93b, KP93a] for more information on the second and third tasks.

In Section 2, we explain how mappings can be used to represent reordering transformations. In Section 3, we describe tuple relations and how they are used to represent mappings and data dependences. In Section 4 we state the legality test for a complete mapping. In Section 5, we present the process of finding legal mappings in terms of defining a search tree. In Sections 6 and 7 give more details on how to construct the search tree. In Section 8, we discuss related work, and finally in Section 9, we give our conclusions.

2 Mappings

2.1 Iteration spaces

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

$$
\begin{array}{llll}
I_{10}: & \{ & [\,i\,] & \mid 1 \le i \le n \; \} \\
I_{20}: & \{ & [\,i,\ j\,] & \mid 1 \le i \le n \wedge 1 \le j \le i-1 \; \} \\
I_{30}: & \{ & [\,i\,] & \mid 1 \le i \le n \; \}
\end{array}
$$

Figure 1: Program and associated iteration space

Each statement s_p has associated with it an iteration space I_p, which is a subspace of $\mathcal{Z}^{m_p}$ (where m_p is the number of loops nested around s_p). A statement's iteration space is the set of iterations for which that statement will be executed. Figure 1 shows a program and its associated iteration space.

2.2 Mappings

We represent reordering transformations as 1-1 mappings from the original iteration spaces (the I_p's), to a new iteration space I'. In general, we associate a separate mapping with each statement. For efficiency reasons, however, it may sometimes be advantageous to partition statements into groups and associate the same mapping with each statement in the same group.

We use the following notation to represent each mapping:

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

where:

- The $i_p^1, \ldots, i_p^{m_p}$ are the index variables of the loops nested around statement s_p.
- The f_p^j's (called *mapping components*) are quasi-affine functions of the iteration variables and symbolic constants. Quasi-affine expressions [AI91] are affine functions plus integer division and remainder when dividing by a constant. Quasi-affine expressions allow us to create mappings corresponding to blocking (or strip-mining) transformations.

This mapping represents the fact that iteration $[i_p^1, \ldots, i_p^m]$ in the original iteration space of statement s_p is mapped to iteration $[f_p^1, \ldots, f_p^n]$ in the new iteration space. For example, the mapping in Figure 2 maps iteration $[5, 7]$ in the original iteration space of statement 20 to iteration $[1, 7, 0, 5]$ in the new iteration space.

$$
\begin{array}{llllll}
T_{10}: & \{[\,i\,] & \rightarrow [0, & i, & 0, & 0]\;\} \\
T_{20}: & \{[\,i,\ j\,] & \rightarrow [1, & j, & 0, & i]\;\} \\
T_{30}: & \{[\,i\,] & \rightarrow [1, & i-1, & 1, & 0]\;\}
\end{array}
$$

```
      parallel do 10 i = 1, n
10      s(i) = 0
      do 30 t = 1, n-1
        parallel do 20 i = t+1, n
20        s(i) = s(i) + a(t,i)*b(t)
30      b(t+1) = b(t+1) - s(t+1)
```

Figure 2: Mapping and associated transformed program

2.3 Code Generation

It is obviously not useful merely to be able to represent transformations as mappings. Given a section of code and a mapping, we need to be able to produce a new section of code that results from applying the transformation represented by that mapping. Since we are only considering reordering transformations, the transformed code will contain the same elementary statements as the original code, but will contain different loop structures. The new loop structures must execute all iterations in the new iteration space and no others. The transformed code also has the important property that all of the iterations are executed in lexicographical order based on their coordinates in the new iteration space. Note that this property specifies a total order on all iterations (even between iterations belonging to different statements), since all iterations in the original iteration spaces are mapped to a common iteration space.

Figure 2 shows the code that is produced by applying the given mapping to the program in Figure 1. Since the `do 10 i ...` and the `do 20 i ...` loops carry no dependencies, they can be run in parallel. The mapping does not specify which loops are made parallel, although given the mapping and dependences, it is easy to determine which loops could be made parallel.

The algorithm we use to generate the transformed code is relatively complicated and is not described in this paper. A description of the algorithm can be found in [KP93b].

2.4 Representing traditional transformations

In [KP93b] we demonstrated how mappings can be used to represent all transformations than can be obtained by applying any sequence of the following traditional transformations:

- 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]

2.5 Examples

Figure 3 gives some interesting examples of mappings. In this paper we don't give details of how to select these particular mappings.

3 Tuple Relations and Sets

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 describe dependences exactly using integer tuple relations. Integer tuple relations are also used to represent mappings.

3.1 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. 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.

Tuple relations and sets are represented using the Omega test [Pug92, PW92, PW93] which is a package for manipulating affine constraints over integer variables. We introduce new variables corresponding to each of the input positions and output positions. Relationships between these variables and those corresponding to symbolic constants are represented as a disjunction of convex regions. See [Pug91] for a more thorough description.

The gist operation

We make use of the gist operation, 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 $q) \wedge q) = (p \wedge q)$), otherwise it is **False**.

3.2 Control dependence

If conditionals exist in a program, then we require that they be converted to guarded statements via if-conversion [AKPW83]. Alternatively, structured if statements can be handled by treating them as atomic statements. We also require that all loop bounds be affine functions of surrounding loop variables and symbolic constants. We can therefore ignore control dependences, as those that do exist are implicitly contained in our description of the iteration space.

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

Code adapted from CHOSOL in the Perfect club (SD)

Original code

```
      do 3  i=2,n
1       sum(i) = 0.
        do 2  j=1,i-1
2         sum(i) = sum(i) + a(j,i)*b(j)
3       b(i) = b(i) - sum(i)
```

Dependences

$$
\begin{array}{lll}
d_{12} : \{[i] & \to [i,l] & | 1 \le l < i \le n\} \\
d_{22} : \{[i,l] & \to [i,l'] & | 1 \le l < l' < i \le n\} \\
d_{23} : \{[i,l] & \to [i] & | 1 \le l < i \le n\} \\
d_{32} : \{[i] & \to [i',i] & | 2 \le i < i' \le n\}
\end{array}
$$

Mapping (for parallelism)

$$
\begin{array}{llllllll}
T_1 : & \{ [\, i \,] & & \to [\, 0, & i, & 0, & 0 \,] \,\} \\
T_2 : & \{ [\, i, & j \,] & \to [\, 1, & j, & 0, & i \,] \,\} \\
T_3 : & \{ [\, i \,] & & \to [\, 1, & i-1, & 1, & 0 \,] \,\}
\end{array}
$$

Transformed code

```
     parallel do 1  i = 2,n
1      sum(i) = 0.
     do 3  t2 = 1, n-1
       parallel do 2  i = t2+1,n
2        sum(i) = sum(i) + a(t2,i)*b(t2)
3      b(t2+1) = b(t2+1) - sum(t2+1)
```

Transformations required normally

- loop distribution
- imperfectly nested triangular loop interchange

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

Original code

```
      do 2  p = 1, n
        do 2  q = 1, p
          do 2  i = 1, orb
1           xrsiq(i,q)=xrsiq(i,q) + ... S1
2           xrsiq(i,p)=xrsiq(i,p) + ... S2
```

Dependences

$$
\begin{array}{lll}
d_{11} : \{[p,q,i] & \to [p',q,i] & | 1 \le q \le p < p' \le n\} \\
d_{12} : \{[p,p,i] & \to [p,p,i] & | 1 \le p \le n\} \\
d_{22} : \{[p,q,i] & \to [p,q',i] & | 1 \le q < q' \le p \le n\} \\
d_{21} : \{[p,q,i] & \to [p',p,i] & | 1 \le q \le p \le p' \le n\}
\end{array}
$$

Schedule (for parallelism)

$$
\begin{array}{llllllll}
T_1 : & \{ [\, p, & q, & i \,] & \to [\, i, & q, & p, & 0]\} \\
T_2 : & \{ [\, p, & q, & i \,] & \to [\, i, & p, & q, & 1]\}
\end{array}
$$

Transformed code

```
     parallel do 12 i = 1,orb
       parallel do 12 t2 = 1,n
         do 21 t3 = 1,t2-1
21         xrsiq(i,t2)=xrsiq(i,t2) + ... S2
11       xrsiq(i,t2)=xrsiq(i,t2) + ...   S1
22       xrsiq(i,t2)=xrsiq(i,t2) + ...   S2
         do 12 t3 = t2+1,n
12         xrsiq(i,t2)=xrsiq(i,t2) + ... S1
```

Transformations required normally

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

Figure 3: Example Codes, Mappings, and Resulting Transformations

3.3 Data dependence

We use tuple relations to represent dependences. If there is a (flow, output or anti) 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 tuple i to tuple j.

3.4 Mappings

If the transformation maps iteration i in the original iteration space of statement s_p to iteration j in the new iteration space, then the mapping T_p will map tuple i to tuple j.

4 Verifying the Legality of Mappings

The previous section showed how reordering transformations can be represented as mappings. Not all transformations are legal, so it is important to be able to distinguish between legal and illegal transformations. In this section we describe how to verify that a mapping corresponds to a legal transformation.

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.

So we have the legality requirement: 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:

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

where $\prec$ is the lexicographically precedes operator. To be well formed, the mapping must also be 1-1.

5 Defining the Search Tree of Legal Mappings

So far we have described how to represent reordering transformations as mappings and how to verify the legality of a given mapping. To make use of such a framework, optimizing compilers would need to be able to select legal mappings that would produce efficient transformed code. In this section we show how this can be modeled as a search problem. More precisely, for a given section of code, we will define a search tree whose leaves correspond to all legal mappings for that section of code. A complete search tree could never actually be constructed by an optimizing compiler as such search trees are all infinitely large. However, by having a well defined search tree, any search algorithm can be used to find a good legal mapping. Some examples of possible search algorithms include:

- Using a heuristic to choose which branch to follow at each fork.
- An interactive tool where a human user can select some or all of the branches to follow at each fork.
- Defining a function from the nodes to the integers which satisfies the monotone property [Nil80]. This would allow us to use admissible search algorithms such as the A^* algorithm [Nil80]. In a previous paper [KP93a], we described an approach along these lines.

Depending on the search algorithm used, parts of the search tree may actually be constructed, or may only exist implicitly. We will not discuss search algorithms further in this paper, but will instead concentrate on defining the search tree.

5.1 Divide and Conquer

In order to define a search tree, we must first decompose the mapping selection problem into a sequence of choices. There are many different ways in which this problem can be decomposed. The decomposition that we have chosen has the desirable property that all partial legal sequences of choices can be extended into a complete legal sequence of choices (i.e., a legal mapping).

Coarse Grained Decomposition

At the highest level, we decompose the mapping selection problem into the smaller problems of choosing each of the mapping components. The sequence in which these choices are made is very important. We require that the mapping components for levels $1, \ldots, k$ of statement s_p be chosen before the mapping component for level $k+1$ of statement s_p. We also prefer[1] that the mappings components for levels $1, \ldots, k$ of all other statements s_q be chosen before the mapping component for level $k+1$ of statement s_p.

As was explained in Section 4, the legality test for complete mappings involves dependence relations and the lexicographically precedes operator. The lexicographically precedes operator is defined as:

$$(x_1, \ldots, x_n) \prec (y_1, \ldots, y_n) \;\text{ iff }\; \exists_m \text{ s.t. } (\forall i \;\; 1 \le i < m \Rightarrow x_i = y_i) \wedge x_m < y_m$$

Therefore, if we are given shorter tuples $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_k)$ such that $(x_1, \ldots, x_k) \preceq (y_1, \ldots, y_k)$ (where $k < n$), then we know that we can always extend them (by choosing $x_{k+1}, \ldots, x_n$ and $y_{k+1}, \ldots, y_n$) such that

$$(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n) \prec (y_1, \ldots, y_k, y_{k+1}, \ldots, y_n)$$

So, given a partially specified mapping (where only the first k levels of the mapping has been chosen), we know that it can be extended into a legal complete mapping iff

$$\forall i, j, p, q, Sym \quad i \rightarrow j \in d_{pq} \;\Rightarrow T_p^k(i) \preceq T_q^k(j) \tag{2}$$

where T_p^k is the mapping consisting of only the first k levels of statement s_p.

[1] If this is not the case then the above desirable property is not guaranteed to hold.

When we are choosing mapping components at level $k+1$, we only need to consider dependences which haven't already been guaranteed to be respected by mapping components chosen at levels $1, \ldots, k$. A dependence $i \rightarrow j \in d_{pq}$ is guaranteed to be respected by the mapping components chosen at levels $1, \ldots, k$, iff $\forall Sym\ T_p^k(i) \prec T_q^k(j)$. So, if Equation 2 has been maintained while selecting mapping components for levels $1, \ldots, k$, then at level $k+1$, we need only consider dependences:

$$d_{pq}^{k+1} : \{i \rightarrow j \text{ s.t. } i \rightarrow j \in d_{pq} \wedge T_p^k(i) = T_q^k(j)\}$$

Fine Grained Decomposition

We distinguish two parts of a mapping component: the *variable part* and the *constant part*. The variable part is the largest subexpression of the mapping component that is a linear function of the iteration variables. The rest of the expression is called the constant part. For example in the mapping $[i, j] \rightarrow [2i + j + n + 1, 0, j]$ the first mapping component has variable part $2i + j$ and constant part $n + 1$.

We further decompose the mapping selection problem by splitting the selection of each mappings component into two parts; selecting the variable part and selecting the constant part. The performance of the resulting program will mostly depend on the selection of the variable parts. The choice of constant parts will affect the legality of the overall mapping, however they will usually have little affect on performance. So, it makes sense to separate the selection of the variable parts from the selection of the constant parts, since the selection of variable parts requires careful consideration, while any legal constant parts will usually suffice.

Although we won't be discussing how to choose good mappings (mappings that will produce efficient code), the techniques we use for characterizing legal mappings could also be used to assist in the selection of good mappings. The efficiency of the produced code depends largely on which loops are parallel which in turn depends on which levels carry dependences. Our techniques for characterizing legal mappings can be modified so as to characterize legal mappings which don't carry any dependences at a given level, or similarly, characterize those that carries all remaining dependences at a given level.

In Section 6, we characterize the set of legal variable parts, and in Section 7, we characterize the set of legal constant parts.

6 Characterizing Legal Variable Parts

In this section we will characterize the sets of legal variable parts for all statements at a given level k. As was mentioned earlier, at level k, we need only consider those dependences d_{pq}^k, that are not carried by levels $1, \ldots, k-1$. For the remainder of this section we will use d_{pq} to denote d_{pq}^k. Unless we are at the last level, we don't need to ensure that all remaining dependences are carried at this level. Rather, we simply need to ensure that they are not violated; that is, if there remains a dependence from iteration i of s_p to iteration j of s_q then we only require that $f_p^k(i)$ be less than *or equal to* $f_q^k(j)$.

We first describe a necessary condition for a variable part v to be legal for statement s_p at level k. We will describe a set V_p for each statement s_p such that if variable part $v \notin V_p$ then v can't be used as a variable part for statement s_p at level k. Depending on which variable parts are chosen for the other statements at this level, a variable part $v \in V_p$ may or may not be legal; that is, inclusion in V_p is necessary but not sufficient for legality.

6.1 Self dependences

A variable part is in V_p only if it respects all self dependences for that statement. A self dependence is a dependence from one iteration of a statement to a different iteration of the same statement. Self dependences come in two forms:

Direct Iteration i of statement s_p is directly dependent on iteration j of statement s_p.

Transitive Iteration i of statement s_p is dependent on iteration x of statement s_q and iteration x of statement s_q is dependent (possibly transitively) on iteration j of statement s_p.

This suggests that we need to compute the transitive closure of the dependence relations (as suggested in [Pug91]). As it turns out, this isn't exactly what we want or need.

Firstly, exact computation of the transitive closure of an affine integer tuple relation is undecidable (see Section 6.2). We can, however, often compute closure of relations exactly, or bound them from above or below. It would be safe to approximate transitive closure from below as this would lead to looser constraints in V_p. This might lead us to select an invalid variable part v_p. This will not lead to invalid mappings, however, because if v_p is invalid we will not be able to find mappings for the other statements such that the tests in Section 4 are satisfied. This would lead to backtracking in the search algorithm.

Secondly, even if a variable part satisfies all self dependences for a particular statement, including transitive dependences, it may be the case that the variable part cannot be used regardless of which variable parts are chosen for the other statements. Figure 4 shows a contrived program and its corresponding dependence graph that demonstrates this situation. Since there are no self dependences for statement 1, even including transitive dependences, any variable part appears legal for statement 1. However, if $-i$ is used as the variable part of the mapping for statement 1, there is no possible affine mapping for statement 2 that would respect the dependences. This surprising situation occurs because we have restricted our variable parts to be linear functions. If we were to relax that criterion, then a legal variable part could be found for statement 2, even if we choose variable part $-i$ for statement 1.

```
      do 2 i = 0, 2
1     a(i) = b(5-2*i)
2     b(i) = a(-i) + b(5-2*i)
```

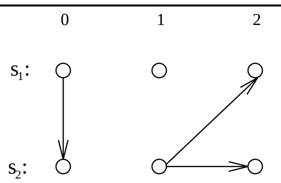


Figure 4: Example program and associated dependence graph

6.2 Affine closure

Since we don't want to consider non linear mappings, we can use this information to develop a stronger test for inclusion in V_p. Consider two iterations i and j of statement s_p such that there is a self dependence from i to j. We say that statement s_p has a *self dependence distance* of $j - i$. We use Δ_p to refer to the set of all self dependence distances of statement s_p. If v is a legal *linear* variable part, then for any two iterations i' and j' of statement s_p separated by a distance of $d \in \Delta_p$, we know that $v(i') \leq v(j')$. So, even if there is no actual data dependence from i' to j', we can "pretend" that there is; that is, adding such a dependence to our dependence set will not reduce the set of legal linear variable parts. We call such dependences *affinity dependences* since they can only be added because we are using affine mappings. The properties of affine mappings also imply that if $d_1 \in \Delta_p \wedge d_2 \in \Delta_p$ then $\alpha_1 d_1 + \alpha_2 d_2 \in \Delta_p$ provided that α_1 and α_2 are nonnegative rationals and $\alpha_1 d_1 + \alpha_2 d_2$ is integral.

For each statement s_p, we replace d_{pp} with the dependences $\{x \rightarrow y \mid x \in [s_p] \wedge y \in [s_p] \wedge \exists d \in \Delta_p \text{ s.t. } x + d = y\}$. We call this the *affine closure* of d_{pp}. In Section 6.3.1, we will explain how the set of all self dependences for a given statement can be "compressed" into a form that can be represented using a single convex set of constraints.

Figure 5 shows the modified dependence graph for the example in Figure 4. Variable part $-i$ is no longer legal for statement 1, so the situation described above no longer occurs in this example. It remains an open question however, whether such situations could arise in other examples despite the introduction of affinity dependences as described above.

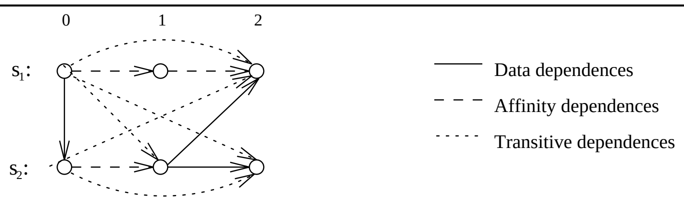


Figure 5: Dependence graph with added dependences

It is important to note that we can only add these affinity dependences between iterations of the same statement. We can't add affinity dependences between statements because the statements will most likely have different mapping components. This does not mean however, that dependences between different statements are not important. To explain how these inter-statement dependences are used we need to briefly review [Pug91] how to compute the transitive closure of an entire dependence graph. Consider the simple case where there are only two statements of interest, s_p and s_q. To generate all transitive self dependences on statement s_p we would compute:

$$(d_{pp} \cup (d_{pq} \circ (d_{qq})^* \circ d_{qp}))^* \tag{3}$$

There are two operators used in this computation, the composition operator and the closure operator. Composition is very easy to compute, but the closure of a relation cannot always be converted to a closed form using only affine constraints. For example, $\{[x,y] \rightarrow [x+1, y+z]\}^*$ is equivalent to $\{[x,y] \rightarrow [x', y+z(x'-x)] \mid x \leq x'\}$ which cannot be represented using only affine constraints. The interesting thing to note about this formula is that the closure operator is only applied to self dependences. This motivates the following approach. Instead of using Equation 3 to generate transitive dependences, we generate affine transitive dependences by computing:

$$(d_{pp} \cup (d_{pq} \circ (d_{qq})^{@} \circ d_{qp}))^{@}$$

where @ is the affine closure operator. This process of computing affine transitive dependences can be extended to the multiple statement case in the obvious way [Pug91].

The affine transitive self-dependences introduced, may produce new self dependence distances, so it might be possible to infer more information by iterating the above process. We don't know of any upper bound on the number of times this process would need to be repeated before reaching a fixed point. We suspect it would be small, and even if it wasn't it would be safe to stop after any number of iterations.

We can efficiently compute an upper bound on the affine closure of the dependences using just direction/distance vectors. This will be approximate only if the direction/distance vectors do not exactly describe the dependence or if all the statements do not have the same nesting depth. This fast approximate computation can save us work in our exact computation. Our exact computation starts with the direct dependences (a lower bound on the affine closure of the dependences), and adds induced dependences step by step. If during the computation of the exact computation, we discover that the exact result computed so far is the same as the approximate result then we can terminate the computation early since no additional dependences can be induced.

6.3 Case I: No variable parts already known

Given all self dependences d_{pp} for a statement s_p, we now wish to give a concise description of V_p. In previous work [KP93b] we used generate and test techniques; that is, for a given variable part v we described a test that determined whether or not $v \in V_p$. The test was simply:

$$\forall d \in \Delta_p \quad v \cdot d \geq 0$$

We would prefer a more concise description of V_p; that is, we would like to describe it in the form:

$$V_p = \{a^1 i_p^1 + \ldots + a^n i_p^n \mid F\}$$

where F is a set of linear constraints on $\{a^1, \ldots, a^n\}$.

There are various techniques we can use to generate F. The most complete, but also most expensive method is to apply the affine form of Farkas Lemma [Sch86, Fea92a]:

Lemma 6.1 (Farkas) *Let the system $Ax \geq b$ of affine inequalities have at least one solution. An affine form ψ is non-negative for each x satisfying $Ax \geq b$ if and only if $\psi(x) \geq 0$ is a nonnegative affine combination of the inequalities in the system $Ax \geq b$.*

The Δ sets are represented internally as a disjunction of convex regions. Each of these convex regions is represented by a system of affine inequalities $A\delta \geq b$. Farkas Lemma is applied to each of these convex regions, with $\psi(\delta) \equiv a^1\delta^1 + \ldots + a^n\delta^n$.

If we only want to consider variable parts that carry all remaining dependences at this level, then we require that $a^1\delta^1 + \ldots + a^n\delta^n > 0$; that is, we use $\psi(\delta) \equiv a^1\delta^1 + \ldots + a^n\delta^n - 1$. Similarly, if we only want to consider variable parts that don't carry any dependences at this level, then we require that $a^1\delta^1 + \ldots + a^n\delta^n = 0$. We would apply Farkas Lemma twice, with $\psi(\delta) \equiv (a^1\delta^1 + \ldots + a^n\delta^n)$ and $\psi(\delta) \equiv -(a^1\delta^1 + \ldots + a^n\delta^n)$.

Each application of Farkas Lemma produces a system of linear inequalities on $a^1, \ldots, a^n$ (Appendix A explains how to apply Farkas Lemma). The set of constraints F is the intersection of each of these sets of constraints.

In practice, it is often the case that each of the systems of affine inequalities describing the Δ sets are very simple. In these cases, we can directly produce constraints on $a^1, \ldots, a^n$ without having to resort to applying Farkas Lemma. For example, if a system of affine inequalities describes a constant self dependence distance, then we can directly produce the constraint $(c^1a^1 + \ldots + c^na^n) \geq 0$ where $\delta^j = c^j$ is known.

Figure 6 gives an example of generating constraints in case I.

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

$$d_{11}: \begin{array}{ll} \{[i,j]->[i,j+1] & \mid 1 \le i \le n \wedge i+1 \le j < n\} \cup \\ \{[i,j]->[i+1,j] & \mid 1 \le i < n \wedge i+1 \le j \le n\} \end{array}$$

$$\Delta_1: \{[0,1]\} \cup \{[1,0\}$$

From the first clause we can directly derive: $0a_1 + 1a_2 \ge 0$

and from the second clause we can directly derive: $1a_1 + 0a_2 \ge 0$

Combining these we get: $V_1 = \{a_1 i_1 + a_2 i_2 \mid a_1 \ge 0 \wedge a_2 \ge 0\}$

Figure 6: Example of deriving V in case I

6.3.1 Compressing self dependence relations

Except in the case where dependences are forced to be carried, all of the constraints on $a^1, \ldots, a^n$ produced in Section 6.3 are guaranteed to be linear, as opposed to just being affine (i.e., their constant terms are alway zero). Assume for the moment that one of these constraints is: $(c^1 a^1 + \ldots + c^n a^n) \ge 0$. If the original program had had an additional dependence distance of $(c^1, \ldots, c^n)$ then the set of constraints on $a^1, \ldots, a^n$ would not be altered; that is, the set of legal variable parts would not be altered by the addition of that dependence distance. We can therefore "pretend" that this self dependence difference exists even if there are no dependences in the original program with that dependence distance. Assume then, that we have added such a self dependence difference for each constraint in the system. If at this point we removed all other self dependence distances, then the same set of constraints on $a^1, \ldots, a^n$ would still be induced. We now have a finite set of self dependence distances $d_1, \ldots, d_m$ which we can use to replace the original dependence distances. Taking the affine closure of this set of dependence distances produces a set of dependence distances which can be described using a single convex set of constraints:

$$\{x \mid \exists \alpha_1, \ldots, \alpha_m \ge 0 \text{ s.t. } x = \alpha_1 d_1 + \ldots + \alpha_m d_m\}$$

From this we can easily construct the self dependence relation which will also be described using a single convex set of constraints. So, given any set of self dependences for a statement, we can compute an equivalent set of dependences (i.e., a set which induces the same constraints on the mapping coefficients) that can be described using a single convex set of constraints.

6.4 Case II: Some variable parts already known

We now consider the situation where we have already chosen variable parts for some of the statements. This additional knowledge allows us to further constrain the set of variable parts which can be used for the other statements. If we have already chosen variable part v_p for statement s_p, then we know that iteration i of statement s_p will be executed no latter than iteration j of statement s_p if $v_p(i) \le v_q(j)$. So, even if there is no actual data dependence from i to j, we can "pretend" that there is. We call such dependences *mapping dependences* since they can only be added because we know something about the mappings used. These newly introduced dependences are not useful in themselves, but they may allow us to transitively infer self dependences for other statements whose variable parts haven't already been chosen.

These additional dependences are useful, but we can do even better. Consider the situation shown in Figure 7(a). We have chosen variable part v_q for statement s_q and are currently considering variable parts for statement s_p. There are actual data dependences from iteration i_1 of statement s_p to iteration j_1 of statement s_q and from iteration j_2 of statement s_q to iteration i_2 of statement s_p. We also assume for this example that $v_q(j_1) \le v_q(j_2)$. By the above reasoning we can infer a mapping dependence from iteration j_1 to iteration j_2 and a transitive dependence from iteration i_1 to iteration i_2. More interesting however, we can also infer that the variable part for statement s_p must be

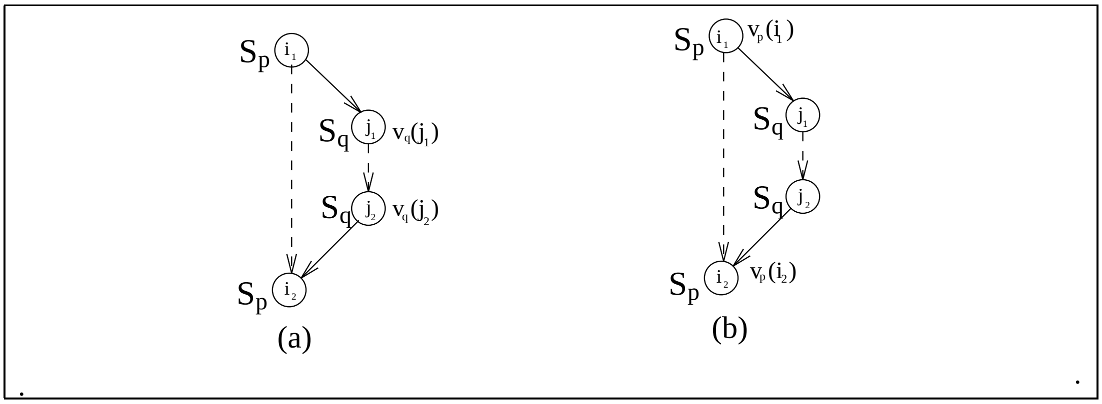
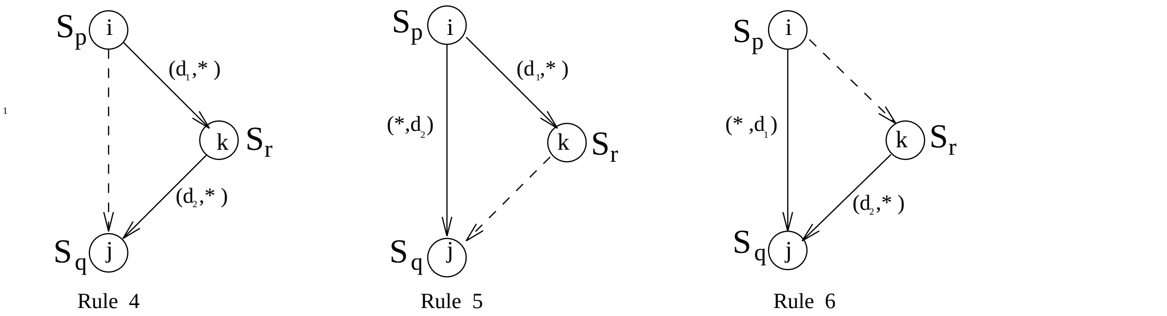


Figure 7: Deriving upper and lower bounds on variable parts



$$
\begin{array}{rcll}
i \rightarrow j \in d_{pq} & \Rightarrow & i \rightarrow j_{(0,*)} \in d_{pq} & (4)\\
\text{Variable part already chosen for } s_p & \Rightarrow & i \rightarrow j_{(v_p(j-i),v_p(j-i))} \in d_{pp} & (5)\\
i \rightarrow k_{(d1,*)} \in d_{pr} \wedge k \rightarrow j_{(d2,*)} \in d_{rq} & \Rightarrow & i \rightarrow j_{(d1+d2,*)} \in d_{pq} & (6)\\
i \rightarrow k_{(*,d1)} \in d_{pr} \wedge i \rightarrow j_{(d2,*)} \in d_{pq} & \Rightarrow & k \rightarrow j_{(*,d1-d2)} \in d_{rq} & (7)\\
i \rightarrow j_{(*,d1)} \in d_{pq} \wedge k \rightarrow j_{(d2,*)} \in d_{rq} & \Rightarrow & i \rightarrow k_{(*,d1-d2)} \in d_{pr} & (8)
\end{array}
$$

Figure 8: Rules for inferring d_{min} and d_{max} values

chosen such that: $v_p(i_2) - v_p(i_1) \geq v_q(j_2) - v_q(j_1)$. The dual of this situation is shown in Figure 7(b). In this example we can infer that the variable part for statement s_q must be chosen such that: $v_p(i_2) - v_p(i_1) \geq v_q(j_2) - v_q(j_1)$.

We generalize this line of reasoning by associating with each dependence, two quantities d_{max} and d_{min}. If a dependence from iteration i of statement s_p to iteration j of statement s_q has associated values d_{max} and d_{min}, then the variable parts must satisfy the following inequalities

$$d_{min} \leq v_p(i) - v_q(j) \leq d_{max}$$

The d_{max} and d_{min} values are derived according to the rules listed in Figure 8. If more than one rule is applicable, then the largest d_{min} value and smallest d_{max} value is used.

These augmented self dependence relations d^*_{pp} are maintained in disjunctive normal form. They involve two additional variables d_{min} and d_{max}. For each convex region we generate two systems of affine inequalities: Δ^{min} on new variables $\delta_1, \ldots, \delta_n$ and d_{min} and Δ^{max} on new variables $\delta_1, \ldots, \delta_n$ and d_{max}. We once again apply Farkas Lemma or one of the simpler techniques described in Section 6.3 to each of these systems of inequalities. This time using $\psi(\delta) \equiv (a^1\delta_1 + \ldots + a_n\delta_n) - d_{min}$ with Δ^{min}, and $\psi(\delta) \equiv d_{max} - (a^1\delta_1 + \ldots + a_n\delta_n)$ with Δ^{max}.

The characterization F of V_p is the intersection of all resulting sets of constraints.

Figure 9 gives an example of this construction process.

```
            do 2 i = 1, n
  1           a(i) = b(i-1) + c(3*i)
  2           b(i) = a(i) + 1
```

Variable part $3i$ already chosen for statement 1.

Actual data dependences:

$d_{12}: \{[i] \rightarrow [i] \mid 1 \leq i \leq n\}$
$d_{21}: \{[i] \rightarrow [i+1] \mid 1 \leq i < n\}$

Transitive dependences:

$d_{11}: \{[i] \rightarrow [i'] \mid 1 \leq i < i' \leq n\}$
$d_{12}: \{[i] \rightarrow [i'] \mid 1 \leq i \leq i' \leq n\}$
$d_{21}: \{[i] \rightarrow [i'] \mid 1 \leq i < i' \leq n\}$
$d_{22}: \{[i] \rightarrow [i'] \mid 1 \leq i < i' \leq n\}$

Augmented dependences implied by rule 4:

$d^*_{11}: \{[i] \rightarrow [i']_{(0,*)} \mid 1 \leq i < i' \leq n\}$
$d^*_{12}: \{[i] \rightarrow [i']_{(0,*)} \mid 1 \leq i <= i' \leq n\}$
$d^*_{21}: \{[i] \rightarrow [i']_{(0,*)} \mid 1 \leq i < i' \leq n\}$
$d^*_{22}: \{[i] \rightarrow [i']_{(0,*)} \mid 1 \leq i < i' \leq n\}$

Augmented dependences implied by rule 5:

$d^*_{11}: \{[i] \rightarrow [i']_{(3i'-3i,3i'-3i)} \mid 1 \leq i < i' \leq n\}$

Augmented dependences implied by rule 6:

$d^*_{12}: \{[i] \rightarrow [i']_{(3i'-3i,*)} \mid 1 \leq i < i' \leq n\}$
$d^*_{21}: \{[i-1] \rightarrow [i']_{(3i'-3i,*)} \mid 2 \leq i < i' \leq n\}$
$d^*_{22}: \{[i-1] \rightarrow [i']_{(3i'-3i,*)} \mid 2 \leq i < i' \leq n\}$

Augmented dependences implied by rules 7 and 8:

$d^*_{12}: \{[i] \rightarrow [i'-1]_{(*,3i'-3i)} \mid 1 \leq i < i' \leq n\}$
$d^*_{21}: \{[i] \rightarrow [i']_{(*,3i'-3i)} \mid 1 \leq i < i' \leq n\}$
$d^*_{22}: \{[i] \rightarrow [i'-1]_{(*,3i'-3i)} \mid 1 \leq i < i' \leq n\}$

Summary of augmented self dependences for statment 2:

$d^*_{22}: \{[i] \rightarrow [i']_{(3i'-3i-3,3i'-3i+3)} \mid 1 \leq i < i' \leq n\}$

$\Delta_2^{min}: \{[\delta_1, d_{min}] \mid \delta_1 > 0 \wedge d_{min} = 3\delta_1 - 3\}$ $\quad \Delta_2^{max}: \{[\delta_1, d_{max}] \mid \delta_1 > 0 \wedge d_{max} = 3\delta_1 + 3\}$

Applying farkas lemma to Δ_2^{min} with $\psi(\delta) \equiv a_1 i_1 - d_{min}$ gives: $a_1 \geq 3$

Applying farkas lemma to Δ_2^{max} with $\psi(\delta) \equiv d_{max} - a_1 i_1$ gives: $a_1 \leq 3$

After combining these constraints we get: $V_2 = \{a_1 i_1 \mid a_1 = 3\}$

Figure 9: Example of deriving V in case II

7 Characterizing Legal Constant Parts

Once we have a legal variable part v_p for each statement s_p, we must, if possible, select constant parts that align the variable parts.

We create a new variable c_p 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 $f_p = v_p + c_p$. 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.

In the usual case, we only need to satisfy all remaining dependences. However, as was explained in Section 5, we sometimes require that the constant parts be chosen so that all remaining dependences are carried at this level or require that no dependences are carried at this level. These requirements can be easily met by making very minor changes to the following formulas (analogous to the corresponding changes made in Section 6.3).

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

$$\forall i, j, Sym\ i \rightarrow j \in d \Rightarrow f_p(i) \leq f_q(j) \tag{9}$$

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

$$\forall i, j \text{ s.t. } i \rightarrow j \in d \Rightarrow v_p(i) + c_p \leq v_q(j) + c_q \tag{10}$$

which is the set of constraints on c_p, c_q and the symbolic constants that are imposed by the simple dependence relation d.

At this point, we could apply Farkas Lemma since Equation 10 is equivalent to requiring that the affine form $v_q(j) + c_q - v_p(i) + c_p$ is nonnegative at all points in the convex region described by d. We prefer however, to use the following methods, as we have found they are more efficient in practice. Our alternative approaches are not guaranteed to produce exact results in all situations, however we haven't encountered any non-contrived programs where they don't produce satisfactory results.

Firstly, we rewrite Equation 10 as:

$$A: \quad \neg(\exists i,j \text{ s.t. } i \rightarrow j \in d \wedge v_p(i) + c_p > v_q(j) + c_q) \tag{11}$$

Unfortunately, the negation in Equation 11 usually produces a disjunction of several constraints. Our goal now, is to deduce from Equation 11 a system of linear constraints for the alignment constants. The conditions, D, under which the dependence exists are:

$$D: \quad \exists i,j \text{ s.t. } i \rightarrow j \in d \tag{12}$$

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 11 is equivalent to:

$$(\exists i,j \text{ s.t. } i \rightarrow j \in d) \Rightarrow \neg(\text{gist } \exists i,j \text{ s.t. } i \rightarrow j \in d \wedge v_p(i) + c_p > v_q(j) + c_q \text{ given } \exists i,j \text{ s.t. } i \rightarrow j \in d) \tag{13}$$

Usually, the gist in Equation 13 will produce a single inequality constraint. If the gist produced a disjunction of inequality constraints, we could strengthen the condition by throwing away all but one of the inequalities produced by the gist.

Unfortunately, Equation 13 also contains an implication operator. So, rather than constructing the set of constraints described by Equation 13, we construct a slightly stronger set of constraints by changing the antecedent to true:

$$\neg(\text{gist } \exists i,j \text{ s.t. } i \rightarrow j \in d \wedge v_p(i) + c_p > v_q(j) + c_q \text{ given } \exists i,j \text{ s.t. } i \rightarrow j \in d) \tag{14}$$

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.

In practice, it is often the case that each of the dependence relations d, are very simple. In some of these cases we can directly produce constraints on the c_p's without resorting to the above gist calculation. For example, if d describes that there is a dependence between all iterations separated by a distance of x in direction i^m, then if both statements have variable part i^m, we can directly produce the constraint $c_p \leq c_q + x$.

For a given pair of statements s_p and s_q, we form a single set of constraints A_{pq} by combining the alignment constraints (Equation 14) resulting from all simple dependence relations between those two statements. We then combine the A_{pq} 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.

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.

8 Related Work

The framework of Unimodular transformations [Ban90, WL91, 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. Most existing frameworks use dependence direction/distance vectors as a dependence abstraction rather than the dependence relations that we use. This is adequate for unimodular frameworks, but is not adequate to test the legality of the sorts of transformations that we can represent. 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. Our mappings are not required to be unimodular and can therefore also generate steps.

Paul Feautrier [Fea92a, Fea92b] has independently developed a framework which is very similar to our own. It is similar in the following respects:

- He represents reordering transformations using schedules which are similar in form to our mappings.
- He generates a separate schedule for each statement.
- We both select mappings/schedules one level at a time, using the dependences that are not carried at outer levels to test legality.

However, we differ from Feautrier in the following respects:

- Unlike our mappings, Feautrier's schedules are not required to be 1-1. Instead, iterations that are to be executed in parallel are scheduled at the same point in time. Therefore, Feautrier's schedules (the time mapping) only partially specify the transformed code. In a separate decision process (the space mapping), parallel loops are generated to enumerate all the computations that need to be executed at each time point. This framework only allows the generation of innermost parallel loops; outer parallel loops are often desirable.
- His methods are designed to generate a schedule that produces code with a "maximal" amount of parallelism. He does this by generating a large set of constraints which describe all legal schedules. This set of constraints has a variable for each coefficient and each constant term of the schedule for each statement. For example, for the code from `olda` in Figure 3, the problem generated by Feautrier would have 6 variables for each statement: 3 each for the coefficients of p, q and i, 2 each for the coefficients of n and orb and 1 each for the constant term. He then introduces two linear functions of these variables, one representing the number of iterations that will be executed sequentially and a second representing how many dependences will be carried. These functions and constraints are then combined and transformed into a dual programming problem that is solved using Parametric Integer Programming (PIP). The net result of this process is that the schedule selected carries as many dependences as possible and among all such schedules, the one selected has as few sequential iterations as possible. These schedules will often not be optimal in practice because of issues such as granularity, data locality and code complexity. It is unclear if his method could be extended to include other criteria, such as good cache performance or parallel outer loops. We expect it would be difficult to encode such an optimization function for a code segment containing several statements.

  Our framework is designed to provide a setting in which multiple performance issues can be traded-off. To satisfy this goal, we select mappings in a completely different way. We start by considering some statement, and generate the constraints on choices for legal variable parts for that statement. We then choose one of these legal variable parts by methods described in [KP93a]. We incorporate that information into the constraints for the variable parts of the other statements. We continue this selection process until we have selected a variable part for each statement. We then move on to making decisions about constant parts in a similar way. Once we have selected mapping components for all statements at the current level we calculate which dependences still remain and repeat this selection process at the next level. This process continues until we have a 1-1 mapping which satisfies all dependences. If not all of decisions made during this process are perfect then backtracking is required if we want to find the optimal transformation. This method differs fundamentally from Feautrier's in that at each stage we are "trying" specific mapping components. Working with actual mapping components, rather than with formulas describing mapping components, makes it much easier to analyze complex performance issues such as data locality.

Feautrier's approach runs into serious problems with large problems. Feautrier reports requiring two hours to generate a schedule for a code fragment with 24 assignment statements. We believe that in many situations, there will be a large number of cases where only a single choice is both legal and reasonable. Also, making a choice for the variable part of one statement may impose strict requirements on the variable parts of other statements in the same strongly connected component of the dependence graph. This would allow our local search approach to be quite efficient. We hope and expect that our methods will allow us to handle large examples efficiently. But of course, we won't know until we complete our re-implementation and integrate it with a decision engine for guiding the local search.

9 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 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 described two basic techniques:

- Techniques that infer constraints on the legal mappings for individual statements or the relationship between the mappings for different statements. Using these methods, we can easily produce legal mappings. Other methods, such as those described in [KP93a], are required to determine which of the possible legal mappings are desirable.
- Techniques that given mappings for some statements can infer constraints on the legal mappings for the other statements. Without these techniques, we would often choose what appears to be a legal mapping for a statement, only to find out later that we can't find compatible mappings for the other statements.

We have described a variety of methods for deriving constraints on the sets of legal mappings. For most examples, applying all of these methods will be overkill. We hope to learn more about which combinations of these methods are cost effective in practice when we complete our implementation. One possibility is to vary the level of effort dynamically, as we encounter dead-ends or situations in which we have many reasonable choices that appear legal. In interactive environments where a user is guiding the optimization choices, a high level of effort is probably warranted.

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.

[AI91] Corinne Ancourt and Franccois Irigoin. Scanning polyhedra with DO loops. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[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.

[DR92] Alain Darte and Yves Robert. Scheduling uniform loop nests. In *Proceedings of ISMN International Conference on Parallel and Distributed Computer Systems*, October 1992. Also available as Technical Report 92-10, Laboratoire de l'Informatique du Parallélisme, Ecolo Normal Supérieure de Lyon, Instiut IMAG.

[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.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II, Multidimensional time. *Int. J. of Parallel Programming*, 21(6), Dec 1992.

[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.

[KP93a] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. Technical Report CS-TR-3108, Dept. of Computer Science, University of Maryland, College Park, July 1993. to appear in Parallel Processing Letters (1994).

[KP93b] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[LMQ91] Herve Leverge, Christophe Mauras, and Patrice Quinton. A language-orientied approach to the design of systolic chips. *Journal of VLSI Signal Processing*, pages 173–182, Mar 1991.

[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.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, 1980.

[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.

[PW93] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

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

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, Great Britain, 1986.

[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] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

[Wol89b] Michael Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, pages 655–664, November 1989.

Appendix A - How to apply Farkas Lemma

We have a set of affine constraints:

$$\begin{array}{rcl} A_{11}x_1 + \ldots + A_{n1}x_n - b_1 & \geq & 0 \\ \vdots & & \\ A_{1m}x_1 + \ldots + A_{nm}x_n - b_m & \geq & 0 \end{array}$$

and an affine form $\psi(x) \equiv a_1x_1 + \ldots + a_nx_n$.

Since we require this affine form to be nonnegative for all x's, it must be a non-negative combination of the affine constraints:

$\exists \lambda_0, \ldots, \lambda_m \geq 0$ s.t. $\forall x_1, x_2, \ldots, x_n$

$$\begin{array}{rcl} a_1x_1 + \ldots + a_nx_n & = & \lambda_0 + \lambda_1(A_{11}x_1 + \ldots + A_{n1}x_n - b_1) + \ldots + \lambda_m(A_{1m}x_1 + \ldots + A_{nm}x_n - b_m) \\ & = & (\lambda_0 - \lambda_1 b_1 + \ldots - \lambda_m b_m) + (A_{11}\lambda_1 + \ldots + A_{1m}\lambda_m)x_1 + \ldots + (A_{n1}\lambda_1 + \ldots + A_{nm}\lambda_m)x_n \end{array}$$

By equating matching coefficients of the x_i's on each side of the equality, we can derive the following linear set of constraints:

$$\exists \lambda_0, \ldots, \lambda_m \geq 0 \text{ s.t. } \begin{array}{rcl} 0 & = & \lambda_0 - \lambda_1 b_1 + \ldots - \lambda_m b_m \\ a_1 & = & A_{11}\lambda_1 + \ldots + A_{1m}\lambda_m \\ & \vdots & \\ a_n & = & A_{n1}\lambda_1 + \ldots + A_{nm}\lambda_m \end{array}$$