Unimodular Transformations of Non-Perfectly Nested Loops

Jingling Xue

Department of Mathematics, Statistics and Computing Science
University of New England
Armidale 2351, Australia

Abstract

A framework is described in which a class of imperfectly nested loops can be restructured using unimodular transformations. In this framework, an imperfect loop nest is converted to a perfect loop nest using Abu-Sufah's Non-Basic-to-Basic-Loop transformation. Conditions for the legality of this transformation and techniques for their verification are discussed. An iteration space, which extends the usual concept so as to represent explicitly the executions of individual statements, is proposed to model the converted loop nest. Since the converted loop nest is a perfect loop nest, data dependences can be extracted and the optimal transformation can be selected for parallelism and/or locality in the normal manner. To generate the restructured code for a unimodular transformation, a code generation method is provided that produces the restructured code that is free of if statements by construction.

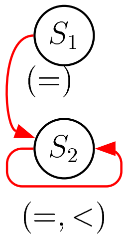
Keywords: Unimodular transformation; Imperfect loop nest; Data dependence; Non-Basic-to-Basic-Loop transformation; Fourier-Motzkin elimination; Feautrier's PIP

1 Introduction

Loop restructuring transformations, such as loop interchange, skewing and reversal, are often applied to loop nests to expose parallelism or otherwise optimise performance. To deal with perfect loop nests systematically, a loop transformation theory, based on unimodular matrix transformations, has been developed [6, 27]. In this framework, a perfect loop nest is represented by the convex polyhedron defined by its loop bounds, known as the iteration space. Loop interchange, skewing, reversal and any combination of the three are modeled as unimodular matrix transformations on the iteration space. A transformation maps the iteration space to another iteration space, called the *transformed iteration space*. In the transformed iteration space, the iterations are executed lexicographically with respect to that space. To generate the restructured code is to construct a (usually perfect) loop nest that enforces this new execution order. This can be carried out using one of several existing methods, including the Fourier-Motzkin elimination [3], Feautrier's parametric integer programming (PIP) [12], the Omega Calculator developed by Pugh and his associates [21, 22], and the algorithm described in [27]. An extension to allow non-unimodular transformations exists [29].

Unfortunately, this unimodular approach is limited to perfect loop nests only. In the case of imperfect loop nests, several techniques have been suggested, including loop distribution [28] and Abu-Sufah's Non-Basic-to-Basic-Loop transformation (abbreviated to NBBL henceforth) [1]. In the absence of dependence cycles, loop distribution can be used to obtain several perfect loop nests, which are then amenable to the unimodular approach in the normal manner. For example, the dependence graph for the loop nest below:

$$\begin{array}{ll} & \textsf{do } I = 1,\ 10 \\ S_1: & \quad C(I) = 0 \qquad (=) \\ & \quad \textsf{do } J = 1,\ 10 \\ S_2: & \qquad A(I, 2J) = A(I, J) + C(I) \\ & \qquad\qquad (=, <) \end{array} \tag{1}$$



does not have any dependence cycles. Furthermore, it has no lexically backward dependence arcs. Hence, the outer loop can be distributed:

$$\begin{array}{ll} & \textsf{do } I = 1,\ 10 \\ S_1: & \quad C(I) = 0 \\ & \textsf{do } I = 1,\ 10 \\ & \quad \textsf{do } J = 1,\ 10 \\ S_2: & \qquad A(I, 2J) = A(I, J) + C(I) \end{array}$$

Now, the last two loops form a 2-deep perfect loop nest. In the unimodular framework, these two loops can be interchanged to bring the vectorisable I loop into the inner position.

Sometimes, all statements in an imperfect loop nest are involved in a strongly connected component. While loop distribution is illegal in this case, Abu-Sufah's NBBL transformation may be used to create one single perfect loop nest, as demonstrated in [26, 28]. This transformation simply moves all imperfectly nested statements to the innermost loop, adding if statements as necessary for correct execution. For example, the loops given in (1) will be converted to a basic loop nest:

$$\begin{array}{l} \textsf{do } I = 1,\ 10 \\ \quad \textsf{do } J = 1,\ 10 \\ \qquad \textsf{if } J = 1 \textsf{ then } C(I) = 0 \\ \qquad A(I, 2J) = A(I, J) + C(I) \end{array}$$

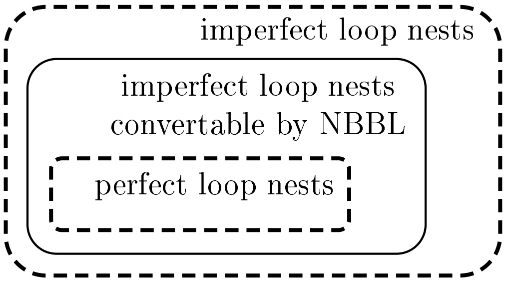


Fig. 1: The scope of this paper's approach indicated with the solid box.

It was suggested [26, 28] that this perfect loop nest be then restructured using the existing unimodular approach. If a loop nest contains if statements, this simple-minded approach will generate the restructured code that contains these if statements (albeit in transformed format). In [26], some heuristics were outlined to eliminate some of the if statements in the restructured code when certain conditions are met. However, these heuristics do not form a systematic approach to eliminating all conditionals. In [26], an important condition for the legality of NBBL was given; but this condition is correct only in the special case of 2-deep loop nests, as will be discussed in Section 3.2.

In this paper, we describe a framework in which a certain class of imperfect loop nests can be restructured using unimodular transformations and guards-free restructured code can be generated systematically. The scope of imperfect loop nests under consideration includes all those amenable to the NBBL transformation, as depicted in Figure 1. Therefore, this work generalises the existing unimodular approach to handle a larger class of loop nests while retaining the philosophy of restructuring code using matrix transformations and polyhedral theories.

In our framework, an imperfect loop nest is first converted to a perfect loop nest using the NBBL transformation. This has the advantage that we can subsequently carry out data dependence analysis and select suitable loop transformations by staying in the existing unimodular framework. In order to generate guards-free restructured code, we introduce an iteration space representation, which extends the usual concept so as to model explicitly the executions of individual statements in a loop nest. We provide a code generation method that takes an iteration space as a union of convex polyhedra and produces the restructured code that is free of if statements by construction. We present conditions for the legality of the NBBL transformation, repairing the deficiency of the existing results. We also discuss efficient techniques for performing the legality test of the NBBL transformation.

A recent study shows that a majority of the loops in scientific code are imperfectly nested, and a majority of the performance-increasing techniques developed in the past two decades assume that loops are perfectly nested [23]. Many important examples are in the form of imperfect loop nests, including LU decomposition without pivoting, Cholesky factorisation and Givens QR factorisation. Therefore, it is of practical importance to develop compiling techniques to deal with loops that are imperfectly nested.

The rest of this paper is organised as follows. Section 2 presents the model of imperfect loop nests we consider in the paper. In Section 3, we explain how to use the NBBL transformation to convert imperfect to perfect loop nests, and discuss the legality of this transformation and techniques for verifying the legality condition. To be self-contained, Section 4 provides pointers to the earlier work on unimodular transformations and data dependence analysis. Section 5 discusses how to generate the restructured code after a unimodular transformation has been applied to a converted loop nest. Section 6 compares this work with related work. Section 7 concludes the paper.

2 Model of Imperfect Loop Nests

The class of imperfect loop nests under consideration in the paper is introduced below (Figure 1). Following [26], an n-deep imperfect loop nest is represented as follows:

$$
\begin{array}{ll}
 & \mathsf{do}\ x_1 = L_1,\, U_1 \\
S_{1a}: & \quad H_{1a}(x_1) \\
 & \quad \mathsf{do}\ x_2 = L_2,\, U_2 \\
S_{2a}: & \qquad H_{2a}(x_1, x_2) \\
 & \qquad\quad \cdots \\
 & \qquad\quad \mathsf{do}\ x_n = L_n,\, U_n \\
S_n: & \qquad\qquad H_n(x_1, \cdots, x_n) \\
 & \qquad\quad \cdots \\
S_{2b}: & \qquad H_{2b}(x_1, x_2) \\
S_{1b}: & \quad H_{1b}(x_1)
\end{array}
\qquad (2)
$$

where the loop bounds of x_k are affine constraints over surrounding loop variables $x_1, \cdots, x_{k-1}$ and some symbolic integer constants, or formally:

$$
\begin{array}{lcl}
L_k & = & \max(\ell_{k,1}, \ell_{k,2}, \cdots) \\
U_k & = & \min(u_{k,2}, u_{k,2}, \cdots)
\end{array}
\qquad (3)
$$

where

$$
\begin{array}{lcl}
\ell_{k,p} & = & \lceil (\ell^0_{k,p} + \ell^1_{k,p} x_1 + \ldots + \ell^{k-1}_{k,p} x_{k-1}) / \ell^k_{k,p} \rceil \\
u_{k,p} & = & \lfloor (u^0_{k,p} + u^1_{k,p} x_1 + \ldots + u^{k-1}_{k,p} x_{k-1}) / u^k_{k,p} \rfloor
\end{array}
\qquad (4)
$$

and all $\ell^q_{k,p}$ and $u^q_{k,p}$ are integer constants, except possibly for $\ell^0_{k,p}$ and $u^0_{k,p}$, which may be symbolic integer constraints but must still be loop invariants in the loop nest. The ceiling and floor functions are introduced to convert rationals to integers. In general, a lower (upper) loop bound is a maximum (minimum) of affine constraints with rational coefficients. This ensures that the space defined by any set of loops in the loop nest is a convex polyhedron. (There exist techniques to handle non-convex iteration space [30]; these techniques allow a loop bound to be an affine expression containing both max and min.)

3 Converting to Perfect Loop Nests

In this section, we explain the conversion of imperfect loop nests to perfect loop nests using Abu-Sufah's Non-Basic-to-Basic-Loop transformation. We discuss the legality of the NBBL transformation and describe techniques for performing the legality test. Finally, we introduce an iteration space representation to model explicitly the executions of individual statements in a loop nest. This representation is the basis for the generation of guards-free restructured code.

3.1 Abu-Sufah's Non-Basic-to-Basic-Loop Transformation

Abu-Sufah's NBBL transformation converts imperfect to perfect loop nests by introducing if statements as necessary for correct execution. By moving all imperfectly nested statements S_{ka} and S_{kb} to the innermost loop, the imperfect loop nest (2) becomes a perfect loop nest:

$$
\begin{array}{ll}
 & \mathsf{do}\ x_1 = L_1,\, U_1 \\
 & \quad \mathsf{do}\ x_2 = L_2,\, U_2 \\
 & \quad\quad \cdots \\
 & \quad\quad\quad \mathsf{do}\ x_n = L_n,\, U_n \\
\overline{S}_{1a}: & \quad\quad\quad\quad \mathsf{if}\ x_2 = L_2 \wedge \cdots \wedge x_n = L_n\ \mathsf{then}\ H_{1a}(x_1) \\
\overline{S}_{2a}: & \quad\quad\quad\quad \mathsf{if}\ x_3 = L_3 \wedge \cdots \wedge x_n = L_n\ \mathsf{then}\ H_{2a}(x_1, x_2) \\
 & \quad\quad\quad\quad \cdots \\
\overline{S}_{n}: & \quad\quad\quad\quad H_n(x_1, \cdots, x_n) \\
 & \quad\quad\quad\quad \cdots \\
\overline{S}_{2b}: & \quad\quad\quad\quad \mathsf{if}\ x_3 = U_3 \wedge \cdots \wedge x_n = U_n\ \mathsf{then}\ H_{2b}(x_1, x_2) \\
\overline{S}_{1b}: & \quad\quad\quad\quad \mathsf{if}\ x_2 = U_2 \wedge \cdots \wedge x_n = U_n\ \mathsf{then}\ H_{1b}(x_1)
\end{array}
\tag{5}
$$

This loop nest is referred to as the *converted loop nest* or *converted program* to distinguish it from the *original loop nest* or *original program* (2). Note that all the statements are identified with $\overline{S}_{ka}$, $\overline{S}_{kb}$ and $\overline{S}_n$ (with overbars) to distinguish them from their counterparts in the original program.

With all statements in the innermost loop being treated as an atomic statement, a perfect loop nest is traditionally represented by the polyhedron defined by the loop bounds,

ITERATIONS IN THE IMPERFECT LOOP NEST (2)	ITERATIONS IN THE CONVERTED LOOP NEST (5)
$S_{ka}[i_1, \cdots, i_k]$	$\overline{S}_{ka}[i_1, \cdots, i_k, L_{k+1}, \cdots, L_n]$
$S_n[i_1, \cdots, i_n]$	$\overline{S}_n[i_1, \cdots, i_n]$
$S_{kb}[i_1, \cdots, i_k]$	$\overline{S}_{kb}[i_1, \cdots, i_k, U_{k+1}, \cdots, U_n]$

Fig. 2: How the iterations in (2) are related to those in (5).

known as the *iteration space*. To generate guards-free restructured code, we shall in Section 3.4 refine the iteration space so that the executions of individual statements are represented explicitly. Before then, we shall use the iteration space in the usual manner.

Some notations used in the following sections are:

- In the imperfect loop nest (2), $S_{ka}[i_1, \cdots, i_k]$ and $S_{kb}[i_1, \cdots, i_k]$ are iterations of S_{ka} and S_{kb}, respectively, when $x_1 = i_1, \cdots, x_k = i_k$. $S_n[i_1, \cdots, i_n]$ is the iteration of S_n when $x_1 = i_1, \cdots, x_n = i_n$.
- In the converted loop nest (5), $\overline{S}_{ka}[i_1, \cdots, i_n]$, $\overline{S}_{kb}[i_1, \cdots, i_n]$ and $\overline{S}_n[i_1, \cdots, i_n]$ are iterations of $\overline{S}_{ka}$, $\overline{S}_{kb}$ and $\overline{S}_n$, respectively, when $x_1 = i_1, \cdots, x_n = i_n$.

Figure 2 shows how the iterations in the original imperfect loop nest are related to those in the converted program. Geometrically, we are attaching the imperfectly nested statements at the boundaries of the iteration space, as illustrated in Figure 3.

3.2 Legality Condition

A loop transformation is *legal* if normal execution of the restructured program will produce the same answer as the original program. This implies that a legal transformation must preserve all dependence relations of the original program.

Unfortunately, the NBBL transformation is not always legal. Figure 4(a) shows an illegal transformation. The iteration $S_1[2]$ in the original program is not executed in the converted program. The problem is that $\overline{S}_1[2, 2]$ is not a valid iteration in the converted program. Consider a more involved example in Figure 4(b). When $I = 4$, the lower bound of the inner loop $\max(-I + 3, I - 1)$ evaluates to 3 and the upper bound to 2. Thus, $S_1[4]$ does not get executed in the inner loop. In others words, $S_1[4, 3]$ is not a valid iteration in the converted program. Hence, the NBBL transformation in the example is illegal. If the upper bound of I were 3 (instead of 4), the NBBL transformation would be legal.

Definition 1 (**Legality of NBBL**) Refer to Figure 2. The NBBL transformation is *legal* if for every iteration $S_{ka}[i_1, \cdots, i_k]$ ($S_{kb}[i_1, \cdots, i_k]$, resp.) in the original imperfect loop nest,

1. A loop nest:

   $$\begin{array}{ll} & \textsf{do } I = 1,\ 3 \\ S_1: & \quad H_1(I) \\ & \quad \textsf{do } J = 1,\ I \\ S_2: & \qquad H_2(I, J) \\ S_3: & \quad H_3(I) \end{array}$$

2. Converted loop nest:

   $$\begin{array}{ll} & \textsf{do } I = 1,\ 3 \\ & \quad \textsf{do } J = 1,\ I \\ \overline{S}_1: & \qquad \textsf{if } J = 1 \textsf{ then } H_1(I) \\ \overline{S}_2: & \qquad H_2(I, J) \\ \overline{S}_3: & \qquad \textsf{if } J = I \textsf{ then } H_3(I) \end{array}$$

3. Iteration space:

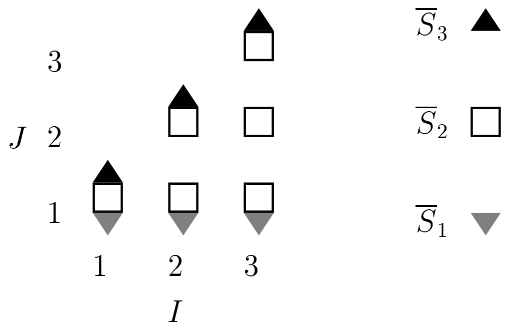


Fig. 3: Abu-Sufah's NBBL transformation.

$\overline{S}_{ka}[i_1, \cdots, i_k, L_{k+1}, \cdots, L_n]$ ($\overline{S}_{kb}[i_1, \cdots, i_k, U_{k+1}, \cdots, U_n]$, resp.) in the converted loop nest is a valid iteration.

This definition ensures that if an iteration of a statement would have executed in the original program, then it still will be in the converted program.

We now formulate the legality condition for applying the NBBL transformation. We first point out that the condition given in [26] is correct only for 2-deep loop nests. To paraphrase [26, p. 49], *the general legality requirement for an n-deep loop nest where S_{ka} or S_{kb} is the outermost imperfectly nested statement is that for each $x_1, \cdots, x_k$ satisfying the k outermost loop bounds, it must be the case that the polyhedron defined by loops x_{k+1} through x_n is nonempty. This ensures that if S_{ka} or S_{kb} would have executed, then it still will.* We use the following example to illustrate why this condition is incorrect and then to

(a)

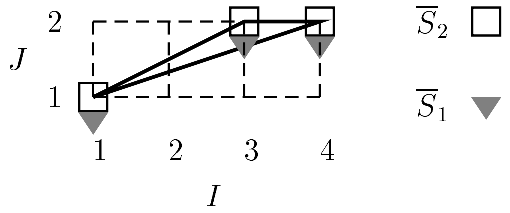
1. A loop nest:

$$
\begin{array}{ll}
 & \texttt{do } I = 1,\, 4 \\
S_1: & \quad H_1(I) \\
 & \quad \texttt{do } J = \lceil (I+2)/3 \rceil,\, \min(\lfloor (I+1)/2 \rfloor, 2) \\
S_2: & \qquad H_2(I, J)
\end{array}
$$

2. Converted loop nest:

$$
\begin{array}{ll}
 & \texttt{do } I = 1,\, 4 \\
 & \quad \texttt{do } J = \lceil (I+2)/3 \rceil,\, \min(\lfloor (I+1)/2 \rfloor, 2) \\
\overline{S}_1: & \qquad \texttt{if } J = \lceil (I+2)/3 \rceil \texttt{ then } H_1(I) \\
\overline{S}_2: & \qquad H_2(I, J)
\end{array}
$$

3. Iteration space:



(b)

1. A loop nest:

$$
\begin{array}{ll}
 & \texttt{do } I = 1,\, 4 \\
S_1: & \quad H_1(I) \\
 & \quad \texttt{do } J = \max(-I+3, I-1),\, 2 \\
S_2: & \qquad H_2(I, J)
\end{array}
$$

2. Converted loop nest:

$$
\begin{array}{ll}
 & \texttt{do } I = 1,\, 4 \\
 & \quad \texttt{do } J = \max(-I+3, I-1),\, 2 \\
\overline{S}_1: & \qquad \texttt{if } J = \max(-I+3, I-1) \texttt{ then } H_1(I) \\
\overline{S}_2: & \qquad H_2(I, J)
\end{array}
$$

3. Iteration space:

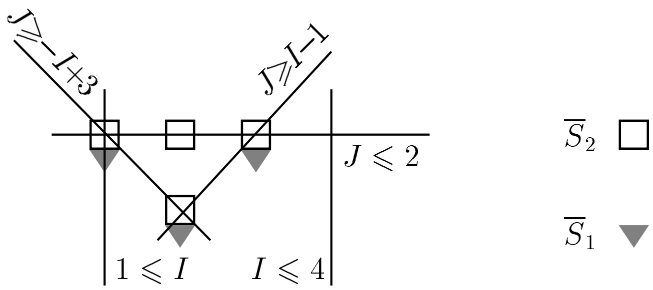


Fig. 4: Two illegal NBBL transformations.

motivate the development of a correct legality condition.

$$
\begin{array}{ll}
 & \mathsf{do}\ I = 1,\ 2 \\
S_1 : & \quad H_1(I) \\
 & \quad \mathsf{do}\ J = 1,\ 2 \\
 & \quad\quad \mathsf{do}\ K = 2,\ J \\
S_3 : & \quad\quad\quad H_3(I, J, K)
\end{array}
$$

Applying the NBBL transformation, we get:

$$
\begin{array}{ll}
 & \mathsf{do}\ I = 1,\ 2 \\
 & \quad \mathsf{do}\ J = 1,\ 2 \\
 & \quad\quad \mathsf{do}\ K = 2,\ J \\
\overline{S}_1 : & \quad\quad\quad \mathsf{if}\ J = 1 \wedge K = 2\ \mathsf{then}\ H_1(I) \\
\overline{S}_3 : & \quad\quad\quad H_3(I, J, K)
\end{array}
\tag{6}
$$

The polyhedron defined by the inner two loops is $\{(J, K) \mid 1 \leqslant J \leqslant 2 \wedge 2 \leqslant K \leqslant J\}$ and contains one point (2,2). Being independent of the outer loop I, this polyhedron is nonempty for each of the two iterations of I. According to the condition in [26] as discussed above, it would be legal to move S_1 into the innermost loop. However, S_1 is executed twice in the original program but never gets executed in the converted program! This is because the guard $J = 1 \wedge K = 2$ is not satisfiable in the entire iteration space.

The problem is that when we move S_1 into the innermost K loop, S_1 passes through the intervening J loop. Thus, we should interpret the transformation as consisting of two moves. Each time, we must not move the statement inwards into a loop unless we are certain that for all iterations in which the statement is executed before the move, that loop executes at least once. In the current example, moving $H_1(I)$ into the J loop yields:

$$
\begin{array}{l}
\mathsf{do}\ I = 1,\ 2 \\
\quad \mathsf{do}\ J = 1,\ 2 \\
\quad\quad \mathsf{if}\ J = 1\ \mathsf{then}\ H_1(I) \\
\quad\quad \mathsf{do}\ K = 2,\ J \\
\quad\quad\quad H_3(I, J, K)
\end{array}
\tag{7}
$$

This transformation is legal, because both operations $H_1(1)$ and $H_1(2)$ that are executed in the original program will be executed in the converted program at $J = 1$. If we next move "$\mathsf{if}\ J = 1\ \mathsf{then}\ H_1(I)$" into the K loop, we will obtain the loop nest (6). This transformation is illegal, because $H_1(1)$ and $H_1(2)$ that are executed at iterations $(1, 1)$ and $(2, 1)$ in the program (7) will not be executed at all in the final program.

We are now ready to state the legality condition for S_{ka} in the case of the n-deep loop nest (2). When moved into the innermost loop x_n, S_{ka} will go through in sequence the intervening loops $x_{k+1}, \cdots, x_{n-1}$. Consider the scenario when S_{ka} is nested inside the x_m

loop and is about to be moved into the x_{m+1} loop, $k \leqslant m < n$:

```
do x_1 = L_1, U_1
  ...
  do x_k = L_k, U_k
    ...
    do x_m = L_m, U_m
      if x_{k+1} = L_{k+1} ∧ ··· ∧ x_m = L_m then H_{ka}(x_1, ···, x_k)
      do x_{m+1} = L_{m+1}, U_{m+1}
        ...
        do x_n = L_n, U_n
          ...
```

All iterations of H_{ka} in the above program can be specified by the following conjunction of affine constraints:

$$P_{ma} \quad = \quad L_1 \leqslant x_1 \leqslant U_1 \wedge \cdots \wedge L_m \leqslant x_m \leqslant U_m \wedge x_{k+1} = L_{k+1} \wedge \cdots \wedge x_m = L_m$$

Lemma 1 *It is legal to move S_{ka} into the innermost loop x_n if and only if*

$$\forall\ k \leqslant m < n, P_{ma} \Longrightarrow L_{m+1} \leqslant U_{m+1}$$

Proof. The transformation of moving S_{ka} into the innermost loop consists of $n-k$ moves, with the process of passing through each loop x_{m+1} along the way counted as one move, $k \leqslant m < n$. In each move, the condition ensures that we do not move S_{ka} into the x_{m+1} loop unless all iterations of S_{ka} before the move, which are specified by P_{ma}, will be executed after the move. Therefore, the NBBL transformation is legal for S_{ka} if and only if the condition stated in this lemma is satisfied. ■

To apply this lemma to the current example, we first check the legality of moving S_1 into the J loop. Since $P_1 = 1 \leqslant I \leqslant 2$, $L_2 = 1$ and $U_2 = 2$, we have $P_1 \Longrightarrow L_2 \leqslant U_2$. So it is legal to obtain the program (7). To further move S_1 into the loop K, we calculate that $P_2 = 1 \leqslant I \leqslant 2 \wedge J = 1$, $L_3 = 2$ and $U_3 = J$. Since $J = 1$, we have $L_3 \nleqslant U_3$. Thus, $P_2 \Longrightarrow L_3 \leqslant U_3$ does not hold. So it is illegal to bring S_1 into the innermost loop K.

A version of this lemma for S_{kb} is stated next. Let P_{mb} be a conjunction of affine constraints that specifies all iterations for S_{kb} when S_{kb} is nested in the x_m loop and is about to be brought into the x_{m+1} loop, $k \leqslant m < n$:

$$P_{mb} \quad = \quad L_1 \leqslant x_1 \leqslant U_1 \wedge \cdots \wedge L_m \leqslant x_m \leqslant U_m \wedge x_{k+1} = U_{k+1} \wedge \cdots \wedge x_m = U_m$$

Lemma 2 *It is legal to move S_{kb} into the innermost loop x_n if and only if*

$$\forall\ k \leqslant m < n,\ P_{mb} \Longrightarrow L_{m+1} \leqslant U_{m+1}$$

3.3 Legality Test

This section discusses techniques for verifying the legality condition of the NBBL transformation for an n-deep loop nest. In its full generality, the legality test is a form of integer programming. We note that the legality of NBBL can be tested, in general, using the Omega Calculator developed by Pugh and his associates [21, 22]. The Omega Calculator contains a set of routines for manipulating Presburger formulas. Presburger formulas are the expressions consisting of affine constraints over integer variables and the usual logical connectives and quantifiers. Let C be a conjunction of affine constraints obtained from the following formula with all containing celling and floor functions, if any, removed:

$$C \stackrel{\text{def}}{=} L_{m+1} \leqslant x_{m+1} \leqslant U_{m+1}$$

(Note that in our loop nest model the ceiling (floor) functions appear only in the lower (upper) bounds.) To check the legality condition in Lemma 1 is to verify if $P_{ma} \Longrightarrow C$ is a tautology or not, which can be determined by the Omega Calculator. We have introduced x_{m+1} by avoiding having to verify $P_{ma} \Longrightarrow (L_{m+1} \leqslant U_{m+1})$, because $L_{m+1} \leqslant U_{m+1}$ may be a non-linear constraint in the presence of ceiling and floor functions and such non-linear constraints are beyond the Omega Calculator. The legality condition in Lemma 2 can be tackled in a similar manner.

In the example shown in Figure 4(a), it is legal to move S_1 into the inner loop if and only if the following formula is a tautology:

$$1 \leqslant I \leqslant 4 \quad \Longrightarrow \quad \lceil (I+2)/3 \rceil \leqslant J \leqslant \min(\lfloor (I+1)/2 \rfloor, 2)$$

With the ceiling, floor and min functions removed, this formula simplifies to:

$$1 \leqslant I \leqslant 4 \quad \Longrightarrow \quad (I+2)/3 \leqslant J \wedge J \leqslant (I+1)/2 \wedge J \leqslant 2$$

Using the Omega Calculator, we establish the falsehood of this formula and conclude that the NBBL in the example is illegal.

In practice, the majority of programs have simple loop bounds [20, 25]. It is therefore possible to avoid integer programming approach all together. If all loop bound constraints in an n-deep loop nest have integer coefficients, i.e., if the denominators $\ell^k_{k,p}$ and $u^k_{k,p}$ in (4) are ± 1, then the ceiling and floor functions there are not necessary. In this case, $L_{m+1} \leqslant U_{m+1}$ in Lemmata 1 and 2 can be rewritten as a conjunction of affine constraints. Then, the legality test can be reduced to a test for the redundancy of loop bound constraints. A constraint in a constraint system is called *redundant* if it is implied by the other constraints in the system. Some techniques for eliminating redundant constraints are described in [9].

Theorem 1 *Assume that all loop bound constraints in (2) have integer coefficients. The legality tests in Lemmata 1 and 2 can be performed by checking if $L_{m+1} \leqslant U_{m+1}$ is redundant in the constraint systems P_{ma} and P_{mb}, respectively.*

In the case of 2-deep perfect loop nests, the legality test can be performed very efficiently. To check the legality of the NBBL transformation in the example shown in Figure 4(b), we test to see if $\max(-I+3, I-1) \leqslant 2$ is redundant with respect to $P_1 = 1 \leqslant I \leqslant 4$. Simplifying $\max(-I+3, I-1) \leqslant 2$ to $1 \leqslant I \leqslant 3$, we reach a negative conclusion. Thus, the NBBL in the example is illegal.

3.4 Refining the Iteration Space Representation

So far, we have used the usual iteration space to represent a perfect loop nest. The iteration space is taken as the convex polyhedron defined by the loop bounds. This definition does not distinguish the statements in the loop body, treating them as an atomic statement. In this section, we refine the existing iteration space concept so that the executions of individual statements in a loop nest are represented explicitly. Our iteration space representation is the basis for the generation of guards-free restructured code.

For notational convenience, an iteration space will be defined as affine constraints built using the usual logic connectives $\wedge$ and $\vee$ rather than as the set of the points that satisfy these affine constraints.

Definition 2 (**Iteration Space, Statement Space and Iteration Vector**) Let $\mathcal{B}$ be a conjunction of the loop bounds of the converted loop nest (5):

$$\mathcal{B} \;=\; L_1 \leqslant x_1 \leqslant U_1 \wedge \cdots \wedge L_n \leqslant x_n \leqslant U_n$$

The *iteration space* of the converted loop nest (5) is defined as $\mathcal{I} = (\bigvee_k \mathcal{I}_{ka}) \vee \mathcal{I}_n \vee (\bigvee_k \mathcal{I}_{kb})$, where

$$\begin{array}{rcl} \mathcal{I}_{ka} & = & \mathcal{B} \wedge \overbrace{x_{k+1} = L_{k+1} \wedge \cdots \wedge x_n = L_n}^{\text{Guard of } \overline{\mathrm{S}}_{\mathrm{ka}}} \wedge\, x_{n+1} = k \\ \mathcal{I}_n & = & \mathcal{B} \wedge x_{n+1} = n \\ \mathcal{I}_{kb} & = & \mathcal{B} \wedge \overbrace{x_{k+1} = U_{k+1} \wedge \cdots \wedge x_n = U_n}^{\text{Guard of } \overline{\mathrm{S}}_{\mathrm{kb}}} \wedge\, x_{n+1} = 2n-k \end{array} \qquad (8)$$

$\mathcal{I}_x$ is called the *statement space* of statement $\overline{S}_x$ in the converted program (5). We have introduced one extra dimension x_{n+1} to represent the lexical scopes of the statements. For example, a 3-deep loop nest consists of the statements $\overline{S}_{1a}$, $\overline{S}_{2a}$, $\overline{S}_3$, $\overline{S}_{2b}$ and $\overline{S}_{1b}$ listed in lexical order ($n = 3$). The corresponding values of x_4 for these five statements will be 1, 2, 3, 4 and 5. It is not difficult to see that the value of x_{n+1} in a statement space also uniquely

Iteration in the converted loop nest (5)	Iteration Vector
$\overline{S}_{ka}[i_1, \cdots, i_k, L_{k+1}, \cdots, L_n]$	$(i_1, \cdots, i_k, L_{k+1}, \cdots, L_n, k)$
$\overline{S}_n[i_1, \cdots, i_n]$	$(i_1, \cdots, i_n, n)$
$\overline{S}_{kb}[i_1, \cdots, i_k, U_{k+1}, \cdots, U_n]$	$(i_1, \cdots, i_k, U_{k+1}, \cdots, U_n, 2n-k)$

Fig. 5: How the iterations in (2) are related to iteration vectors.

identifies the statement with which the statement space is associated. Figure 5 shows how the iterations in the converted program are identified by their *iteration vectors*.

If the NBBL transformation is legal, the iteration space provides a precise representation for the original imperfect loop nest. All iterations in the original program are identified by the corresponding iteration vectors indicated by composing the tables in Figures 2 and 5. Therefore, in a sequential execution of the converted loop nest, all iterations in the original program are executed in lexicographic order according to their associated iteration vectors. If the NBBL transformation is legal, $\mathcal{B}$ in both $\mathcal{I}_{ka}$ and $\mathcal{I}_{kb}$ of (8) can be simplified to:

$$\mathcal{B} \quad = \quad L_1 \leqslant x_1 \leqslant U_1 \wedge \cdots \wedge L_k \leqslant x_k \leqslant U_k$$

because $L_{k+1} \leqslant x_{k+1} \leqslant U_{k+1} \wedge \cdots \wedge L_n \leqslant x_n \leqslant U_n$ is redundant.

Next, we describe how to put the iteration space in a form suitable for loop transformations and subsequent code generation. In (3), we have defined the scope of allowable loop bounds in a loop nest. Based on this, we can see that the statement space $\mathcal{I}_n$ for the perfectly nested statement $\overline{S}_n$ is a convex polyhedron. But the statement spaces $\mathcal{I}_{ka}$ and $\mathcal{I}_{kb}$ for all imperfectly nested statements $\overline{S}_{ka}$ and $\overline{S}_{kb}$ are possibly not convex polyhedra due to the presence of the guards of $\overline{S}_{ka}$ and $\overline{S}_{kb}$ in (8). It is shown next that both $\mathcal{I}_{ka}$ and $\mathcal{I}_{kb}$ each specify a union of a finite number of convex polyhedra. Hence, the iteration space $\mathcal{I}$ is a union of a finite number of convex polyhedra.

In $\mathcal{I}_{ka}$, every equality $x_i = L_i$ can be rewritten to:

$$L_i \leqslant x_i \leqslant L_i$$

Similarly, every $x_i = U_i$ in $\mathcal{I}_{kb}$ is equivalent to:

$$U_i \leqslant x_i \leqslant U_i$$

Using the notation (3), we have:

$$\begin{array}{rcl}
L_i \leqslant x_i & \equiv & \ell_{i,1} \leqslant x_i \wedge \ell_{i,2} \leqslant x_i \wedge \cdots \\
x_i \leqslant L_i & \equiv & x_i \leqslant \ell_{i,1} \vee x_i \leqslant \ell_{i,2} \vee \cdots \\
U_i \leqslant x_i & \equiv & u_{i,1} \leqslant x_i \vee u_{i,2} \leqslant x_i \vee \cdots \\
x_i \leqslant U_i & \equiv & x_i \leqslant u_{i,1} \wedge x_i \leqslant u_{i,2} \wedge \cdots
\end{array}$$

where both sides of the same equation are equivalent. Hence, the constraints that specify every statement space $\mathcal{I}_{ka}$ or $\mathcal{I}_{kb}$ can always be put into the disjunctive normal form. This implies that every statement space is a union of a finite number of convex polyhedra.

4 Unimodular Transformations

The loops in a converted loop nest are perfectly nested. To apply the unimodular approach, we must analyse the data dependences in the program and select a suitable transformation that preserves these dependences while at the same time optimising a given goal (e.g., parallelism or locality). Techniques for extracting data dependences from a loop nest are well-known [5, 13, 20, 21, 28, 31]. The data dependences for the converted loop nest can be extracted in the normal manner, except that the conditionals in the if statements should be taken into account. Methods useful for finding optimal transformations are discussed in [6, 14, 15, 17, 27].

5 Code Generation

Once a transformation has been applied to an n-deep loop nest, we need to generate the restructured code. Let $T \in \mathbb{Z}^{n\times n}$ be a unimodular transformation:

$$\begin{bmatrix} T & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ x_{n+1} \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \\ y_{n+1} \end{bmatrix} \tag{9}$$

This means that a transformation never alters the relative order of the statements in the body of the converted loop nest. Therefore, instead of $\begin{bmatrix} T & 0 \\ 0 & 1 \end{bmatrix}$, we shall speak of the transformation T. T maps the iteration space $\mathcal{I} = (\bigvee_k \mathcal{I}_{ka}) \vee \mathcal{I}_n \vee (\bigvee_k \mathcal{I}_{kb})$ into another iteration space, known as the *transformed iteration space* and denoted $\mathcal{I}' = (\bigvee_k \mathcal{I}'_{ka}) \vee \mathcal{I}'_n \vee (\bigvee_k \mathcal{I}'_{kb})$. Like the iteration space $\mathcal{I}$, the transformed iteration space $\mathcal{I}'$ is also a union of convex polyhedra. The problem we address in this section is to generate a set of possibly imperfectly nested loops to iterate over the points in $\mathcal{I}'$ in lexicographic order.

Figure 6 illustrates our code generation approach. The iteration space for the example is $\mathcal{I} = \mathcal{I}_1 \cup \mathcal{I}_2$, where

$$\begin{aligned} \mathcal{I}_1 &= 1 \leqslant I \leqslant 4 \wedge 1 \leqslant J \leqslant 1 \wedge K = 1 \\ \mathcal{I}_2 &= 1 \leqslant I \leqslant 4 \wedge 1 \leqslant J \leqslant 3 \wedge K = 2 \end{aligned}$$

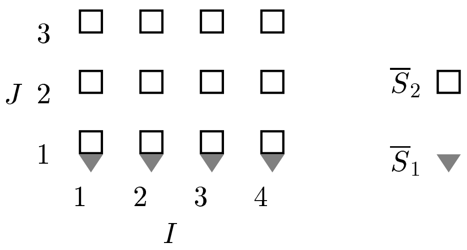
1. A loop nest:

$\quad\quad \texttt{do } I = 1,\ 4$
$S_1: \quad H_1(I)$
$\quad\quad\quad \texttt{do } J = 1,\ 3$
$S_2: \quad\quad H_2(I, J)$

2. Converted loop nest:

$\quad\quad \texttt{do } I = 1,\ 4$
$\quad\quad\quad \texttt{do } J = 1,\ 3$
$\overline{S}_1: \quad\quad \texttt{if } J = 1 \texttt{ then } H_1(I)$
$\overline{S}_2: \quad\quad H_2(I, J)$

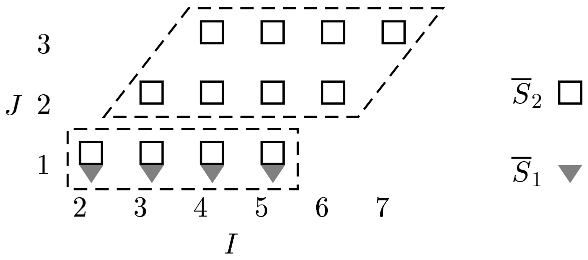
3. Iteration space of the converted loop nest:



4. Transformation:

$$T \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} I' \\ J' \end{bmatrix} \quad T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

5. Transformed Iteration space:



6. Transformed loop nest:

$\texttt{do } I' = 2,\ 7$
$\quad \texttt{do } J' = \max(1, I' - 4),\ 1$
$\quad\quad H_1(I' - J')$
$\quad\quad H_2(I' - J', J')$
$\quad \texttt{do } J' = \max(2, I' - 4),\ \min(I' - 1, 3)$
$\quad\quad H_2(I' - J', J')$

Fig. 6: Unimodular transformation of an imperfect loop nest.

where K represents the lexical scopes of statements $\overline{S}_1$ and $\overline{S}_2$. Applying the transformation in the example, we get the transformed iteration space $\mathcal{I}' = \mathcal{I}'_1 \cup \mathcal{I}'_2$, where

$$\begin{array}{rcl} \mathcal{I}'_1 & = & 1 \leqslant I' - J' \leqslant 4 \wedge 1 \leqslant J' \leqslant 1 \wedge K' = 1 \\ \mathcal{I}'_2 & = & 1 \leqslant I' - J' \leqslant 4 \wedge 1 \leqslant J' \leqslant 3 \wedge K' = 2 \end{array} \tag{10}$$

The restructured code in the example will enumerate the iterations in $\mathcal{I}'$ in lexicographic order. By construction, the restructured code is free of if statements. In the transformed loop nest, each statement is surrounded by n loops. If two statements are enclosed by the same set of loops, they have identical statement spaces. All distinct statement spaces must partition the transformed iteration space. Otherwise, there must be an iteration that is either not enumerated at all or enumerated more than once. In this example, there are two distinct statement spaces:

$$\left\{ \begin{array}{r} 2 \leqslant I' \leqslant 7 \\ \max(1, I' - 4) \leqslant J' \leqslant 1 \end{array} \right. \qquad \left\{ \begin{array}{r} 2 \leqslant I' \leqslant 7 \\ \max(2, I' - 4) \leqslant J' \leqslant \min(I' - 1, 3) \end{array} \right.$$

The first statement space is for the first two statements in the restructured code, and the second for the last statement. In Figure 6, these two statement spaces correspond to the dashed rectangle and parallelogram, respectively. Therefore, code generation for the transformed iteration space can be understood geometrically as a process of partitioning that space into a number of convex polyhedra.

5.1 Preliminaries

According to the transformation (9), the restructured code will consist of loops with y_1 as the first loop variable, and with y_2 as the second loop variable, and so on. When generating code to scan the transformed iteration space $\mathcal{I}'$, every affine constraint in $\mathcal{I}'$ gives a loop bound. Precisely, an affine constraint is a loop bound for the loop variable with the highest index. Let $a_0 + a_1 y_1 + \cdots + a_k y_k \geqslant 0$ be a generic affine constraint such that $a_k \neq 0$. It is a lower bound of y_k if $a_k > 0$ or an upper bound of y_k if $a_k < 0$.

While every affine constraint in $\mathcal{I}'$ always gives a loop bound for some variable, the converse is not true. That is, a given loop variable may not have any of the affine constraints in $\mathcal{I}'$ as an explicit loop bound. This often complicates the code generation process. We illustrate this below by generating code for a single convex polyhedron and provide the rationale for understanding our code generation algorithm.

A convex polyhedron can always be specified by a conjunction of affine constraints. It is always possible to generate a perfect loop nest to scan a convex polyhedron [3, 10]. As

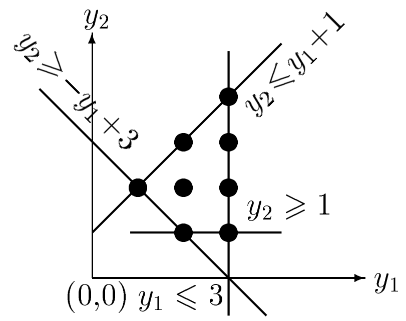


Fig. 7: Code generation for a convex polyhedron.

an example, the following constraints define the convex polyhedron shown in Figure 7:

$$\begin{array}{rcl} -y_1 + 3 & \geqslant & 0 \\ y_1 - y_2 + 1 & \geqslant & 0 \\ y_2 - 1 & \geqslant & 0 \\ y_1 + y_2 - 3 & \geqslant & 0 \end{array}$$

In this example, none of the four constraints is a lower bound of y_1. Previous work indicates that we can use the Fourier-Motzkin elimination or Feautrier's PIP to construct these missing loop bounds. Using the Fourier-Motzkin elimination, we project all the constraints onto the first loop variable y_1 and obtain a lower bound for y_1: $y_1 \geqslant 1$. So the following code will correctly scan the given convex polyhedron:

```
do y1 = 1, 3
   do y2 = max(-y1 + 3, 1), y1 + 1
      Enumerate (y1, y2)
```

In this section, we need to generate code to scan the transformed iteration space $\mathcal{I}' = (\bigvee_k \mathcal{I}'_{ka}) \vee \mathcal{I}'_n \vee (\bigvee_k \mathcal{I}'_{kb})$, which is the union of all transformed statement spaces. When all statements are not executed over a common convex iteration space, we are no longer able to use a perfect loop nest to scan $\mathcal{I}'$. Realising that several transformed statement spaces may overlap, we may interleave the execution of different statements. With the y_{n+1} dimension ignored, we will partition $\mathcal{I}'$ into convex polyhedra (i.e., blocks) such that each block will be enumerated by a perfect loop nest and in every such a loop nest, a statement is either not executed at all or executed at all iterations of the loop nest. We will also structure the loop nests for all partitioning blocks in such a way that the iterations in $\mathcal{I}'$ are enumerated in lexicographic order. Conceptually, it is easy to find one such a partition: two iterations are in the same partitioning block if and only if for every affine constraint in $\mathcal{I}'$, either they both satisfy the affine constraint or they both do not. Based on this, we will partition the example in Figure 6 into two blocks as indicated.

We have developed a recursive algorithm that allows us to generate code by means of finding a partition of $\mathcal{I}'$ incrementally (from outermost to the innermost). By generating code incrementally, we are also able to eliminate redundant constraints incrementally. This simplifies loop bounds and avoids generating too many loops unnecessarily. Let P be a convex polyhedron specified by a conjunction of affine constraints. A constraint c in P is redundant if it is implied by the other constraints in P. This can be tested by checking if (P with c removed) $\wedge \neg c = \emptyset$. If P is a disjunction of conjunctive constraints, we will only attempt to remove redundant constraints in each conjunct separately.

5.2 Algorithm

Given the transformed iteration space $\mathcal{I}'$, we generate code to enumerate the points in $\mathcal{I}'$ in lexicographic order. In the transformed code, the outermost loop iterates over the values of the first loop variable y_1, the one immediate inside the outermost loop iterates over the values of the second loop variable y_2, and so on. We generate the code recursively, level by level, starting at the outermost level for y_1. Our code generation algorithm is given in Figure 8. All formal parameters that specify constraints assume that the constraints are in the disjunctive normal form. The algorithm is explained and analysed below and then illustrated by an example.

Function `GenMain`$(\mathcal{I}')$ is the main program. We first mark all constraints in $\mathcal{I}'$ to indicate that they have not been selected as loop bounds yet. We then call `GenLoopNest` to generate code, starting at the outermost level y_1. The first n levels represent the n loop variables $y_1, \cdots, y_n$ and the last level (i.e., y_{n+1}) indicates the lexical scopes of statements.

Function `GenLoopNest`$(P, O, level)$ generates code recursively to scan P, level by level, starting at $level = 1$. When `GenLoopNest`$(P, O, level)$ is called, the $level - 1$ outermost loops $y_1, \cdots, y_{level-1}$ have already been generated for scanning the first $level - 1$ dimensions of P, and the corresponding loop bound constraints have been saved in the second parameter O. The first parameter P contains affine constraints from which the loop bounds for the remaining dimensions will be generated. We will avoid generating a loop bound constraint if it is already implied by the earlier ones, i.e., those in O (lines 43 — 46). We will elaborate on this using an example in Section 5.4. When returning to the caller, `GenLoopNest`$(P, O, level)$ will have generated the inner loops controlled by $y_{level}, \cdots, y_n$ for scanning the last $n - level + 1$ dimensions of P. In general, these $n - level + 1$ inner loops are not perfectly nested. Thus, several loops at the same depth k may be controlled by the same loop variable y_k, $level \leqslant k \leqslant n$.

When working at a loop level $level$, $1 \leqslant level \leqslant n$, we choose arbitrarily from P a constraint h — if any — that is a loop bound of y_{level} but has not been marked as such in

```
GenMain(I′)   /* Calls GenLoopNest, starting at level=1 */
   for (each constraint c in I′) do
      LoopBound(c) := false /* mark all constraints as not loop bounds */
   return GenLoopNest(I′, true, 1)

GenLoopNest(P, O, level)   /* Generates code, level by level */
   P: the region for which we generate loops on y_level, ···, y_n
   O: loop bound constraints on y_1, ···, y_{level−1} already generated for scanning P
   ElimRedundancies(P, O)
   if Empty(P) then return (∅)
   if (P has some loop bounds of y_level such that LoopBound(h)=false) then
      Select one such a constraint h arbitrarily
      In P ∧ h: LoopBound(h) := true /* mark h as a loop bound */
      L_1 := GenLoopNest(P ∧ h, O, level)
      In P ∧ ¬h: LoopBound(¬h) := true /* mark ¬h as a loop bound */
      L_2 := GenLoopNest(P ∧ ¬h, O, level)
      if (h is an upper bound) then return (L_1 ⊞ L_2) else return (L_2 ⊞ L_1)
   else
      Loop := GenLoop(P, level)
      if (level = n) then LoopBody := GenStatements(P)
      else LoopBody := GenLoopNest(P, O ∧ Loop, level + 1)
      return (Loop ⊞ LoopBody)

GenStatements(P) /* Generates statements B (initially ∅) represented by y_{n+1} */
   for k := 1 to 2n − 1 do
      if (P contains a constraint y_{n+1} = k) then
         if (k < n) then B := B ⊞ H_{ka}(y_1, ···, y_k)
         else if (k = n) then B := B ⊞ H_n(y_1, ···, y_n)
         else B := B ⊞ H_{(2n−k)b}(y_1, ···, y_k)
   return (B)

GenLoop(P, level)   /* Generates a loop for y_level */
   if (P does not contain a lower or upper bound constraint for y_level) then
      Generate the missing loop bound(s) from the convex hull of P using,
      for example, the Fourier-Motzkin elimination or Feautrier's PIP
   Let y_level ⩾ ℓ_{level,1}, y_level ⩾ ℓ_{level,2}, ··· be all lower bound constraints in P
   Let y_level ⩽ u_{level,1}, y_level ⩽ u_{level,2}, ··· be all upper bound constraints in P
   return do y_level = max(ℓ_{level,1}, ℓ_{level,2}, ···), min(u_{level,1}, u_{level,2}, ···)

ElimRedundancies(P, O)   /* Eliminates redundant constraints in P */
   for (each conjunct C in P) do if Empty(C) then Delete C from P
   for (each conjunct C in P) do
      for (each constraint c in C such that LoopBound(c) = false) do
         Delete c from C if c is redundant in (C ∧ O)
      for (each constraint c in C such that LoopBound(c) = true) do
         Delete c from C if c is redundant in (C ∧ O)
```

Fig. 8: Code generation algorithm (⊞ means sequential composition of two code segments).

earlier recursive calls. We use h to split P into two disjoint regions: $P \wedge h$ in the half space h and $P \wedge \neg h$ in the opposite half space $\neg h$. We mark h and $\neg h$ as loop bounds of y_{level} in the two regions $P \wedge h$ and $P \wedge \neg h$, respectively. If h is an upper bound, $P \wedge h$ should be enumerated before $P \wedge \neg h$; otherwise, the order should be reversed (line 17). Note that the symbol $\boxplus$ in line 17 and the rest of the algorithm denotes sequential composition of two code segments. This produces the desired result so that all points in P are enumerated in lexicographic order.

We repeat this splitting process on both regions. When all constraints of y_{level} have been marked as loop bounds, we call `GenLoop`$(P, level)$ to generate the loop of y_{level} to scan P. Sometimes, y_{level} is not bounded explicitly from above or below by any constraints in P. In this case, we will use, for example, the Fourier-Motzkin algorithm or Feautrier's PIP to generate such missing loop bound(s) from the convex hull of P. The Fourier-Motzkin algorithm [24] generates the loop bound for y_{level} by a succession of eliminations of variables $y_n, \cdots, y_{level+1}$ in that order. This algorithm often performs redundant elimination operations when used to generate different loop bounds. Experiments [7] showed that Feautrier's PIP overcomes this limitation and allows fairly large problems to be solved. If Feautrier's PIP is used, the first $level-1$ loop variables $y_1, \cdots, y_{level}$ will be treated as parameters [10]. When constructing the convex hull of P, we note that P is a disjunction of conjuncts. We first build the convex hull for each conjunct and then generate from these convex hulls the convex hull for P.

If we have generated all the loops for scanning P, we generate the loop body for P by calling `GenStatements`(P). Otherwise, we call `GenLoopNest` recursively to generate loops of $y_{level+1}$ at the next level. `GenStatements`(P) generates the statements represented by the constraints on y_{n+1} contained in P. It is a simple matter of mapping the values of y_{n+1} back to the statements they represent, according to Definition 2. If there are multiple statements, we generate them in lexical order (line 25).

`ElimRedundancies` is an important component of the algorithm. In `GenLoopNest`$(P, O, level)$, we use a constraint h to split P into two disjoint regions: $P \wedge h$ in the half space h and $P \wedge \neg h$ in the opposite half space $\neg h$. Recall that P is a disjunction of conjunctive constraints, representing a union of convex polyhedra. It is possible that a convex polyhedron in P is contained entirely in one of the two half spaces. Thus, the intersection of this convex polyhedron with the opposite half space is empty. In the extreme case, all convex polyhedra in P can be contained in the half space h. When this happens, the region $P \wedge \neg h$ is empty. When a convex polyhedron in P is partly contained in one of the half spaces, some specifying constraints for the polyhedron may become redundant. We have written a procedure `ElimRedundancies`(P, O) so that it is the first statement to be executed in

`GenLoopNest` (line 9). This procedure will detect all the cases mentioned above and remove redundant constraints as desired. The last two for loops need a bit of explanation. To eliminate redundant constraints in a conjunct, we first remove those that have *not* been selected as loop bounds followed by those that have. This prevents us from having to choose identical constraints h more than once or a redundant constraint in line 12. When eliminating redundant constraints in P, we also take into account the loop bound constraints generated for the surrounding loops, contained in O. If all constraints in O are obtained from P in lines 36 — 37, then O does not contain any extra information not present in P. In this case, O in lines 43 — 46 can be ignored. On the other hand, if O contains some constraints generated in lines 33 — 35, then these constraints together with those in P may allow us to remove more redundant constraints in P that would not be eliminated otherwise. In other words, we attempt to eliminate a loop bound constraint as long as it is implied by the bounds of the surrounding loops. Section 5.4 uses an example to illustrate this.

5.3 Analysis

Our algorithm generates code recursively, level by level, starting from the outermost level. Its correctness can be proved by induction on the number of levels and is sketched as follows. When working at a given *level*, `GenLoopNest`$(P, O, level)$ splits the region P to be scanned into two disjoint sections and then calls itself recursively to generate two code segments to scan the two sections, respectively. The correctness of the algorithm follows from two facts. First, when combining the two code segments sequentially, line 17 ensures that the right segment of the two is executed first. Second, when called from within `GenLoopNest`, each of the three procedures `GenStatements`, `GenLoop` and `ElimRedundancies` performs the intended task correctly, as was described in Section 5.2.

The code generation algorithm has an exponential time complexity, because the dominant operation — the Fourier-Motzkin elimination or Feautrier's PIP — in the procedure `ElimRedundancies` is exponential. A preliminary implementation of the algorithm (using the Fourier-Motzkin elimination) shows that the algorithm is practically efficient for practical loop nests, which tend to have a few loops and a few imperfectly nested statements. This is consistent with the results reported earlier on the efficiency of the Fourier-Motzkin elimination [4, 21] and Feautrier's PIP [7, 10].

5.4 Example

To understand our code generation algorithm, we trace its actions when it generates code for the transformed iteration space $\mathcal{I}'$ in Figure 6. The affine constraints that specify $\mathcal{I}'$

was given in (10) and is duplicated here for convenience:

$$\begin{array}{lcl} \mathcal{I}'_1 & = & 1 \leqslant I' - J' \leqslant 4 \wedge 1 \leqslant J' \leqslant 1 \wedge K' = 1 \\ \mathcal{I}'_2 & = & 1 \leqslant I' - J' \leqslant 4 \wedge 1 \leqslant J' \leqslant 3 \wedge K' = 2 \end{array}$$

To begin with, in $\texttt{GenMain}(\mathcal{I}')$, we mark all constraints as having not been selected as loop bounds. We then call $\texttt{GenLoopNest}(\mathcal{I}', \texttt{true}, 1)$ to generate code starting at the outermost level, i.e., at I'.

Since $\mathcal{I}'$ contains no loop bound constraints for I', $\texttt{GenLoop}(\mathcal{I}', 1)$ is called immediately to generate an outermost loop on I'. Using the Fourier-Motzkin elimination on the convex hull of P, we obtain the lower and upper bounds $2 \leqslant I' \leqslant 7$. In this two-dimensional case, a visual inspection of Figure 6 can verify that these constraints are indeed loop bounds for I'. Therefore "$\texttt{do}\ I' = 2,\ 7$" is generated. Then a recursive call $\texttt{GenLoopNest}(\mathcal{I}', 2 \leqslant I' \leqslant 7, 2)$ is created to generate loops for J'. The first parameter $\mathcal{I}'$ contains several loop bound constraints for J', let us assume arbitrarily that $h := J' \leqslant 1$ is selected in line 12. This constraint splits $\mathcal{I}'$ into two disjoint regions $\mathcal{I}' \wedge J' \leqslant 1$ and $\mathcal{I}' \wedge J' \geqslant 2$. We then call the same function recursively on both regions. Below we trace the execution of $\texttt{GenLoopNest}$ on $\mathcal{I}' \wedge J' \leqslant 1$ only.

In the call $\texttt{GenLoopNest}(\mathcal{I}' \wedge J' \leqslant 1, 2 \leqslant I' \leqslant 7, 2)$ (line 14), we set the first parameter P of $\texttt{GenLoopNest}$ to: $P_1 := \mathcal{I}' \wedge J' \leqslant 1$. (The subscripts in P are used to distinguish P's in different recursive calls.) A call to $\texttt{ElimRedundancies}$ simplifies P_1 to:

$$P_1 = \overbrace{I' - J' \leqslant 4 \wedge 1 \leqslant J' \wedge \underline{J' \leqslant 1} \wedge K' = 1}^{\mathcal{I}'_1} \bigvee \overbrace{I' - J' \leqslant 4 \wedge 1 \leqslant J' \wedge \underline{J' \leqslant 1} \wedge K' = 2}^{\mathcal{I}'_2}$$

where the underlined constraints indicate those that have been marked as loop bounds. Note that if we did not use the information in O in $\texttt{ElimRedundancies}$ (i.e., $2 \leqslant I' \leqslant 7$), we would not have been able to remove from P_1 the redundant constraint $1 \leqslant I' - J'$. Note that $2 \leqslant I' \wedge J' \leqslant 1 \Longrightarrow 1 \leqslant I' - J'$. Now, there are two constraints left to choose as h: $I' - J' \leqslant 4$ and $1 \leqslant J'$. It is not difficult to see that each constraint contains entirely both $\mathcal{I}'_1$ and $\mathcal{I}'_2$. Let us choose arbitrarily $h := 1 \leqslant J'$. $\texttt{GenLoopNest}(P_1 \wedge \neg(1 \leqslant J'), 2 \leqslant I' \leqslant 7, 2)$ does not generate any code because $P_1 \wedge \neg(1 \leqslant J')$ defines an empty region. In the call $\texttt{GenLoopNest}(P_1 \wedge 1 \leqslant J', 2 \leqslant I' \leqslant 7, 2)$, we generate code for the following region:

$$P_2 = \overbrace{I' - J' \leqslant 4 \wedge \underline{1 \leqslant J'} \wedge \underline{J' \leqslant 1} \wedge K' = 1}^{\mathcal{I}'_1} \bigvee \overbrace{I' - J' \leqslant 4 \wedge \underline{1 \leqslant J'} \wedge \underline{J' \leqslant 1} \wedge K' = 2}^{\mathcal{I}'_2}$$

This time, we choose $h := I' - J' \leqslant 4$, the only constraint that has not been marked as a loop bound. $P_2 \wedge \neg(I' - J' \leqslant 4)$ defines an empty region. In the call $\texttt{GenLoopNest}(P_2 \wedge I' - J' \leqslant$

$4, 2 \leqslant I' \leqslant 7,\ 2)$, we have

$$P_3 \;= \overbrace{\underline{I' - J' \leqslant 4} \wedge \underline{1 \leqslant J'} \wedge \underline{J' \leqslant 1} \wedge K' = 1}^{\mathcal{I}'_1} \bigvee \overbrace{\underline{I' - J' \leqslant 4} \wedge \underline{1 \leqslant J'} \wedge \underline{J' \leqslant 1} \wedge K' = 2}^{\mathcal{I}'_2}$$

Having identified all loop bound constraints for J', we call $\texttt{GenLoop}(P_3, 2)$ to generate the loop "$\texttt{do}\ J' = \max(1, I' - 4),\ 1$". Finally, we call $\texttt{GenStatements}(P_3)$ to generate the two statements $H_1(I' - J')$ and $H_2(I' - J', J')$ nested in the J' loop.

The generation of the other loops can be traced in a similar manner.

6 Related Work

6.1 Loop Transformations

We can distinguish two approachs to restructuring imperfect loop nests. One approach attempts to restructure an imperfect loop nest *directly* [2, 14, 15, 16, 19]. Feautrier [14, 15] allowed different statements in the loop nest to be transformed by possibly different affine transformations (called mappings), and he has algorithms to find optimal piece-wise affine mappings for those computations that can be completed in linear time. Kelly and Pugh [16] developed a framework in which a variety of transformations such as unimodular tansformations, loop fusion and loop fission can be modelled as affine mappings. However, we note that the problem of how to choose a combination of multiple mappings to exploit collectively the available parallelism in a loop nest in the general case has not been resolved. Lim and Lam [19] described a parallelisation algorithm that uses affine mappings to transform an imperfect loop nest into the communication-free SPMD code.

The other approach converts imperfect to perfect loop nests and then applies the existing unimodular transformation theory or its extended versions [1, 23, 26]. Loop distribution can be used as a pre-processing step to convert an imperfect loop nest into as many loop nests as there are strongly connected components in the dependence graph of the original program. Then the existing unimodular approach can be used on those loops that are perfectly nested.

There are many ways of converting an imperfect loop nest to a perfect loop nest. Sass and Mutka [23] used scalar forward substitution followed by loop distribution. As they pointed out, this approach is ineffective for those consisting of recurrence and complex dependences. Alternatively, one may obtain a perfect loop nest by moving all imperfectly nested statements into the innermost loop by introducing conditionals. This technique, known as Abu-Sufah's non-basic-to-basic transformation [1], has been used in this paper. Once a perfect loop nest is obtained, the existing unimodular approach was suggested to be used. Using the transformation in Figure 6 on the example in the same figure, this simple-minded approach will yield the code with a conditional test:

$$
\begin{array}{lll}
 & \mathsf{do}\ I' = 2,\ 7 & \\
 & \quad \mathsf{do}\ J' = \max(1, I'-4),\ \max(I'-1, 3) & \\
\overline{S}_1: & \quad\quad \mathsf{if}\ J' = 1\ \mathsf{then}\ H_1(I'-J') & \\
\overline{S}_2: & \quad\quad H_2(I'-J', J') &
\end{array}
\tag{11}
$$

In his PhD thesis [26], Wolf gave a condition for the legality of the NBBL transformation, which was shown to be correct only for 2-deep loop nests. Wolf also discussed some heuristics in an attempt to remove some of the if statements when certain conditions are met. However, these heuristics do not form a systematic approach to eliminating all the conditionals in the restructured code.

6.2 Code Generation

In compiling for parallel machines, it is often necessary to construct a set of loops to enumerate a region of points in lexicographic order.

6.2.1 Scanning One Single Convex Polyhedron

When restructuring a perfect loop nest with a convex iteration space, we need to generate code for a convex polyhedron, possibly intersected by a lattice. Let $\{x \mid Ax \leqslant b\}$ be a convex iteration space. Let T be a non-singular transformation $Tx = y$. The transformed iteration space is $\{x \mid AT^{-1}y \leqslant b\} \cap \mathcal{L}(T)$, where $\mathcal{L}(T) = \{u \mid u = Tv \wedge v \in \mathbb{Z}^n\}$ is the lattice defined by the columns of T. If T is unimodular, then $\mathcal{L}(T) = \mathbb{Z}^n$. In this case, the transformed iteration space is also a convex polyhedron. Several methods can be used to generate code to scan a convex polyhedron. Ancount and Irigoin [3], Feautrier's PIP [12] and Wolf and Lam [27] generated perfect loop nests with loop bounds involving max and min functions. Instead, Chamsky [8], Cosnard and Loi [11] and Kelly and Pugh [18] generated loop nests that are not necessarily perfectly nested but with simpler loop bounds. The objectives of these recent methods are to produce loop bounds as simple as possible and to avoid as much as possible executing iterations of outer loops for which no iterations of inner loop are executed.

6.2.2 Scanning Multiple Convex Polyhedra

In the last decade, several methods have been proposed to generate code to scan a non-convex region [8, 11, 16, 18, 30]. In [30], we generalised the class of perfect loop nests that can be parallelised by allowing both max and min to appear simultaneously in a common loop bound. The iteration space of such a loop nest was shown to be a union of convex polyhedra. Under a non-singular transformation, the transformed iteration space was shown to be the intersection of a lattice with a union of convex polyhedra. We described a method to generate a set of possibly imperfectly nested loops to scan such a non-convex region. Since both max and min are allowed in a loop bound, we represented a union of convex polyhedra by a conjunction of disjunctive constraints rather than a disjunction of conjunctive constraints as we did in this paper. The advantage is that, as an example, the following constraints

$$y_2 \geqslant 1 \wedge y_2 \geqslant 1 - y_1 \wedge (y_3 \geqslant 3 \vee y_2 \geqslant (5 - y_1)/2)$$

can be directly expressed as a compound lower bound for y_2:

$$y_2 \quad \geqslant \quad \max(1, 1 - y_1, \min(3, \lceil (5 - y_1)/2 \rceil))$$

In [16, 18], Kelly and Pugh allowed different statements to be transformed by distinct non-singular transformations. Therefore, it is necessary to generate code to scan $\bigcup_k (C_k \cap L_k)$, where C_K is a convex polyhedron and L_k is a lattice. Kelly and Pugh described a code generation algorithm to scan such a region and subsequently implemented the algorithm in their Omega Calculator. In this paper, the code generation problem we have dealt with is to generate code to scan a union of convex polyhedra. Instead of representing a union of convex polyhedra by a conjunction of disjunctive constraints as we did in [30], we represented it as a disjunction of conjunctive constraints as in [16]. With the example from Section 5.4 as input, the Omega Calculator (version 1.0) generates the following code:

```
do I' = 2, 7
   if I' <= 5 then
      H1(I' - 1)
   if I' <= 5 then
      H2(I' - 1, 1)
   do J' = max(2, I' - 4), min(I' - 1, 3)
      H2(I' - J', J')
```

It is unclear whether the Omega Calculator can suppress the generation of the if statements.

7 Conclusion

In the case of imperfect loop nests, some researchers suggested to use the Non-Basic-to-Basic-Loop transformation to create one single perfect loop nest and then apply the existing unimodular approach. One major limitation of this simple approach is that the restructured code consists of expensive if statements in the innermost loop. In this paper, we improved and refined this approach by providing a framework in which a certain class of imperfect loop nests can be restructured using unimodular transformations. We used the Non-Basic-to-Basic-Loop transformation to convert imperfect to perfect loop nests. The advantage of this transformation is that the previous work on data dependence analysis and selecting suitable transformations can be utilised. Unfortunately, the Non-Basic-to-Basic-Loop transformation is not always legal. We described the legality condition of this transformation and discussed techniques for verifying the legality condition, repairing the deficiencies of the existing results. We refined the usual iteration space so that the executions of individual statements in a loop nest are represented explicitly. We showed that the refined iteration space is a union of convex polyhedra. Therefore, the transformed iteration space for a unimodular transformation is also a union of convex polyhedra. We described a method that generates code that is guaranteed to be free of if statements by construction.

The results described here can be directly generalised in two directions. By using the techniques in [29], we can allow non-unimodular transformations. We can also accept loop nests with non-convex iteration spaces, which arise due to the presence of both max and min in the same loop bound. As shown in [30], the iteration space for such a loop nest can still be specified as a union of convex polyhedra.

8 Acknowledgements

I wish to thank all the referees for their many constructive and insightful comments and suggestions, which have greatly improved the presentation of the paper. This work is supported by an Australian Research Council Grant A49600987.

References

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, University of Illinois at Urbana-Champaign, Nov. 1978.

[2] S. P. Amarasinghe, J. M. Anderson, M. S.Lam, and A. W. Lim. An overview of a compiler for a scalable parallel machines. In *Proc. of the 7th Workshop on Languages and Compilers for Parallel Computing*, pages 253–272, Aug. 1993.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *PROc. of the 3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM Press, Apr. 1991.

[4] S. P. Armarasinghe and M. S. Lam. Communication optimisation and code generation for distributed memory machines. In *Proc. of the ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation*, Jun. 1993.

[5] U. Banerjee. *Dependence Analysis for Supercomputing*. Engineering and Computer Science: Parallel Processing and Fifth Generation Computing. Kluwer Academic Publishers, 1988.

[6] U. Banerjee. Unimodular transformations of double loops. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, chapter 10, pages 192–219. MIT Press, 1991.

[7] Z. Chamski. Fast and efficient generation of loop bounds. In D. Trystram, editor, *Proc. of Hawaii Int. Conf. on Parallel Computing*. Elsevier Science Publishers (North Holland), Sept. 1993.

[8] Z. Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. In *Proc. of the 27th Int. Conf. on System Sciences*, volume Part II: Software Technology, pages 14–22, 1994.

[9] M .C. Cheng. General criteria for redundant and nonredundant linear inequalities. *J. of Optimization Theory and Applications*, 53(1):37–42, Apr. 1987.

[10] J.-F. Collard, T. Risset, and P. Feautrier. Construction of DO loops from systems of affine constraints. Technical Report 93-15, Ecole Normale Supérieure de Lyon, May. 1993.

[11] M. Cosnard and M. Loi. A simple algorithm for the generation of efficient loop bounds. *Int. J. of Parallel Programming*, 24(3):265–289, 1996.

[12] P. Feautrier. Parametric integer programming. *Operations Research*, 22:243–268, 1988.

[13] P. Feautrier. Dataflow analysis for array and scalar references. *Int. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[14] P. Feautrier. Some efficient solutions to the affine scheduling problem, Part I, one-dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, Oct. 1992.

[15] P. Feautrier. Some efficient solutions to the affine scheduling problem, Part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992.

[16] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS–TR–2995, Department of Computer Science, University of Maryland, Aug. 1993.

[17] W. Kelly and W. Pugh. Selecting affine mappings based on performance estimations. *Parallel Processing Letters*, 4:205–220, Aug. 1994.

[18] W. Kelly and W. Pugh. A framework for iteration reordering transformations. In *Proc. of the 1st Int. Conf. on Algorithms and Architectures for Parallel Processing (ICA^{3}PP)*, pages 153–612. IEEE Computer Society Press, Apr. 1995.

[19] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *Proc. of the 7th Workshop on Languages and Compilers for Parallel Computing*, pages 92–106, Aug. 1994.

[20] D.E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Proc. of the SIGPLAN 1991 PLDI*, pages 1–14, 1991.

[21] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, 35(8):102–114, Aug. 1992.

[22] W. Pugh and D. Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-3191, Department of Computer Science, University of Maryland, Dec. 1992. Earlier version at SIGPLAN PLDI's 92 conference.

[23] R. Sass and M. Mutka. Enabling unimodular transformations. In *Supercomputing '94*. IEEE Computer Society Press, 1994.

[24] A. Schrijver. *Theory of Linear and Integer Programming.* Series in Discrete Mathematics. John Wiley & Sons, 1986.

[25] Z. Shen, Z. Li, and P. C. Yew. An empirical study on array subscripts and data dependences. In *Proc. of Int. Conf. on Parallel Processing*, volume 2, pages 145–150, Aug. 1989.

[26] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Stanford University, Mar. 1992.

[27] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[28] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[29] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[30] J. Xue. Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, 1996.

[31] Y. Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations. In *Proc. of the 7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca, Aug 1994.