UMIACS-TR-95-48 April, 1994
CS-TR-3457

Transitive Closure of Infinite Graphs and its Applications

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

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

Evan Rosser
ejr@cs.umd.edu
Dept. of Computer Science

Tatiana Shpeisman
murka@cs.umd.edu
Dept. of Computer Science

Univ. of Maryland, College Park, MD 20742

Abstract

Integer tuple relations can concisely summarize many types of information gathered from analysis of scientific codes. For example they can be used to precisely describe which iterations of a statement are data dependent of which other iterations. It is generally not possible to represent these tuple relations by enumerating the related pairs of tuples. For example, it is impossible to enumerate the related pairs of tuples in the relation $\{[i] \to [i+2] \mid 1 \leq i \leq n-2\}$. *Even when it is possible to enumerate the related pairs of tuples, such as for the relation* $\{[i,j] \to [i',j'] \mid 1 \leq i,j,i',j' \leq 100\}$, *it is often not practical to do so. We instead use a closed form description by specifying a predicate consisting of affine constraints on the related pairs of tuples. As we just saw, these affine constraints can be parameterized, so what we are really describing are infinite families of relations (or graphs). Many of our applications of tuple relations rely heavily on an operation called transitive closure. Computing the transitive closure of these "infinite graphs" is very different from the traditional problem of computing the transitive closure of a graph whose edges can be enumerated. For example, the transitive closure of the first relation above is the relation* $\{[i] \to [i'] \mid \exists \beta \text{ s.t. } i'-i = 2\beta \wedge 1 \leq i \leq i' \leq n\}$. *As we will prove, this computation is not computable in the general case. We have developed algorithms that produce exact results in most commonly occurring cases and produce upper or lower bounds (as necessary) in the other cases. This paper will describe our algorithms for computing transitive closure and some of its applications such as determining which inter-processor synchronizations are redundant.*

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

Transitive Closure of Infinite Graphs and its Applications

Wayne Kelly, William Pugh, Evan Rosser and Tatiana Shpeisman

Department of Computer Science
University of Maryland, College Park, MD 20742
{wak,pugh,ejr,murka}@cs.umd.edu

Abstract. *Integer tuple relations can concisely summarize many types of information gathered from analysis of scientific codes. For example they can be used to precisely describe which iterations of a statement are data dependent of which other iterations. It is generally not possible to represent these tuple relations by enumerating the related pairs of tuples. For example, it is impossible to enumerate the related pairs of tuples in the relation* $\{[i] \rightarrow [i+2] \mid 1 \leq i \leq n-2\}$*. Even when it is possible to enumerate the related pairs of tuples, such as for the relation* $\{[i,j] \rightarrow [i',j'] \mid 1 \leq i,j,i',j' \leq 100\}$*, it is often not practical to do so. We instead use a closed form description by specifying a predicate consisting of affine constraints on the related pairs of tuples. As we just saw, these affine constraints can be parameterized, so what we are really describing are infinite families of relations (or graphs). Many of our applications of tuple relations rely heavily on an operation called transitive closure. Computing the transitive closure of these "infinite graphs" is very different from the traditional problem of computing the transitive closure of a graph whose edges can be enumerated. For example, the transitive closure of the first relation above is the relation* $\{[i] \rightarrow [i'] \mid \exists \beta \text{ s.t. } i'-i = 2\beta \wedge 1 \leq i \leq i' \leq n\}$*. As we will prove, this computation is not computable in the general case. We have developed algorithms that produce exact results in most commonly occurring cases and produce upper or lower bounds (as necessary) in the other cases. This paper will describe our algorithms for computing transitive closure and some of its applications such as determining which inter-processor synchronizations are redundant.*

1 Introduction

An *integer tuple relation* is a relation whose domain consists of integer k-tuples and whose range consists of integer k'-tuples, for some fixed k and k'. An integer k-tuple is simply a point in $\mathcal{Z}^k$. The following is an example of a relation from 1-tuples to 2-tuples:

$$\{[i] \rightarrow [i',j'] \mid 1 \leq i = i' = j' \leq n\}$$

These relations can concisely summarize many kinds of information gathered from analysis of scientific codes. For example, the relation given above describes

```
      do 2 i = 1, n
1        a(i,i) = 0
         do 2 j = 1, i
2           b(i,j) = b(i,j) + a(i,j)
```

Fig. 1. Example program

the data dependences from statement 1 to statement 2 in the program shown in Figure 1.

We use the term *dependence relation* rather than tuple relation when they describe data dependences. A dependence relation is a much more powerful abstraction that the traditional dependence distance or direction abstractions. The above program has dependence distance (0), but that doesn't tell us that only the last iteration of j loop is involved in the dependence. This type of additional information is crucial for determining the legality of a number of advanced transformations [3]. Tuple relations can also be used to represent other forms of ordering constraints between iterations that don't necessarily correspond to data dependences. For example, we can construct relations that represent which iterations will be executed before which other iterations. We will see later how these relations can be used to avoid redundant synchronization of iterations executing on different processors. As a third application of relations, we show how they can be used to compute closed form expressions for induction variables.

The next section describes the general form of the relations that we can handle, and the operations that we can perform on them. The remainder of the paper examines the transitive closure operation. First, we describe how transitive closure of relations leads to simple and elegant solutions to several program analysis problems. We then describe the algorithms we use to compute transitive closure.

2 Tuple Relations

The class of scientific codes that is amenable to exact analysis generally consists of `for` loops with affine loop bounds, whose bodies consist of accesses to scalars and arrays with affine subscripts. The following general form of an integer tuple relation is therefore expressive enough to represent most information derived during the analysis of such programs:

$$\{[s_1, \ldots, s_k] \rightarrow [t_1, \ldots, t_{k'}] \mid \bigvee_{i=1}^{n} \exists \alpha_{i1}, \ldots, \alpha_{im_i} \text{ s.t. } F_i \}$$

where the F_i's are conjunctions of affine equalities and inequalities on the input variables $s_1 \ldots, s_k$, the output variables $t_1, \ldots, t_{k'}$, the existentially quantified variables $\alpha_{i1}, \ldots, \alpha_{im_i}$ and symbolic constants. These relations can be written equivalently as the union of a number of simpler relations, each of which can be

operation	Description	Definition
$F \cap G$	Intersection of F and G	$x \rightarrow y \in F \cap G \Leftrightarrow x \rightarrow y \in F \wedge x \rightarrow y \in G$
$F \cup G$	Union of F and G	$x \rightarrow y \in F \cap G \Leftrightarrow x \rightarrow y \in F \vee x \rightarrow y \in G$
$F - G$	Difference of F and G	$x \rightarrow y \in F - G \Leftrightarrow x \rightarrow y \in F \wedge x \rightarrow y \notin G$
$range(F)$	Range of F	$y \in range(F) \Leftrightarrow \exists x$ s.t. $x \rightarrow y \in F$
$domain(F)$	Domain of F	$x \in domain(F) \Leftrightarrow \exists y$ s.t. $x \rightarrow y \in F$
$F \times G$	Cross product of F and G	$x \rightarrow y \in (F \times G) \Leftrightarrow x \in F \wedge y \in G$
$F \circ G$	Composition of F and G	$x \rightarrow z \in F \circ G \Leftrightarrow \exists y$ s.t. $y \rightarrow z \in F \wedge x \rightarrow y \in G$
$F \bullet G$	Join of F and G	$x \rightarrow y \in (F \bullet G) \Leftrightarrow x \rightarrow y \in (G \circ F)$
$F \subseteq G$	F is subset of G	$x \rightarrow y \in F \Rightarrow x \rightarrow y \in G$

Table 1. Operations on tuple relations

described using a single conjunct:

$$\bigcup_{i=1}^{n} \{[s_1, \ldots, s_k] \rightarrow [t_1, \ldots, t_{k'}] \mid \exists \alpha_{i1}, \ldots, \alpha_{im_i} \text{ s.t. } F_i \ \}$$

Table 1 gives a brief description of some of the operations on integer tuple relations that we have implemented and use in our applications. The implementation of these operations is described elsewhere [2] (see also http://www.cs.umd.ed u/projects/omega or ftp://ftp.cs.umd.edu/pub/omega)

In addition to these operations we have also implemented and use in our applications the *transitive closure* operator:

$$x \rightarrow z \in F^* \Leftrightarrow x = z \vee \exists y \text{ s.t. } x \rightarrow y \in F \wedge y \rightarrow z \in F^*$$

and *positive transitive closure* operator:

$$x \rightarrow z \in F^+ \Leftrightarrow x \rightarrow z \in F \vee \exists y \text{ s.t. } x \rightarrow y \in F \wedge y \rightarrow z \in F^+$$

In previous work[4], we developed algorithms for a closely related operation called affine closure. Affine closure is well suited to testing the legality of reordering transformations and is generally easier to compute than transitive closure. But many of our applications require the full generality of transitive closure.

Unfortunately, the exact transitive closure of an affine integer tuple relation may not be affine. In fact, we can encode multiplication using transitive closure:

$$\{[x, y] \rightarrow [x+1, y+z]\}^* \text{ is equivalent to: } \{[x, y] \rightarrow [x', y + z(x' - x)] \mid x \leq x'\}$$

Adding multiplication to the supported operations allows us to pose undecidable questions. Transitive closure is therefore not computable in the general case.

3 Applications

This section describes a number of applications of tuple relations and demonstrates the importance of the transitive closure operator.

Original program:

```
do i = 1, 3
  do j = 1, 4
    a(i,j)=a(i-1,j)+a(i,j-1)+a(i-1,j-1)
```

Program with posts and waits inserted:

```
doacross i = 1, 3
   doacross j = 1, 4
      if (1<i) wait(1,i-1,j)
      if (1<j) wait(2,i,j-1)
      if (1<i and 1<j) wait(3,i-1,j-1)
      a(i,j)=a(i-1,j)+a(i,j-1)+a(i-1,j-1)
      if (i<3) post(1,i,j)
      if (j<4) post(2,i,j)
      if (i<3 and j<4) post(3,i,j)
```

Dependence pattern:

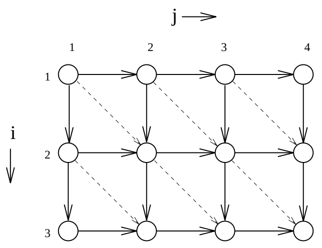

Fig. 2. Example of redundant synchronization

3.1 Simple redundant synchronization removal

A common approach to executing scientific programs on parallel machines is to distribute the iterations of the program across the processors. If there are no dependences between iterations executing on different processors then the processors can execute completely independently. Otherwise, the processors will have to synchronize at certain points to preserve the original sequential semantics of the program. On a shared memory system, the simplest way to achieve the necessary synchronization is to place a `post` statement after the source of each dependence and a corresponding `wait` statement before the sink of each dependence. Figure 2 shows the results of inserting `posts` and `waits` for the given example. As this example demonstrates, and is often the case, many of the posts and waits inserted by this approach are redundant. In this example, we can see that the explicit synchronization that results from the dependence from the write of `a(i,j)` to the read of `a(i-1,j-1)` is redundant, since the appropriate execution ordering will always be achieved due to a chain of explicit synchronizations that result from the other two dependences.

The problem then is to identify which dependences need to be explicitly synchronized. In this section, we restrict ourselves to a simple case of this problem where: the loops are perfectly nested, the granularity of synchronization is between entire iterations of the loop body (i.e., all `posts` occur at the end of the loop body and all `waits` occur at the start of the loop body), and we assume each iteration may execute on a different processor. This is the class of problems considered by some related work [5] in this area. We will show how our approach improves on the related work in this limited domain, then in Section 3.3, we will show how to extend the approach to the more general problem.

We first compute a dependence relation d that represents the data dependences between different iterations of the loop body (see Figure 3 for an example). Each of these dependences will have to be synchronized either explicitly or implicitly. The transitive closure, d^+, of this relation will contain all pairs of iterations that are linked by a chain of synchronizations of length one or more.

```
doacross i ...
  doacross j ...
    a(i+3,j)=b(i-1,j-1)+ ...
    b(i,j) = ...
    ... = b(i-2,j+1)+c(i-1,j-1)
    c(i,j) = a(i,j) + z(i,j)
```

$$\begin{aligned}
d &= \{[i,j] \to [i+3,j]\} \cup \{[i,j] \to [i+2,j-1]\} \cup \{[i,j] \to [i+1,j+1]\} \\
d^{2+} &= \{[i,j] \to [i', j-i+i'-3] \mid i \le i'-3\} \cup \\
&\quad \{[i,j] \to [i',j'] \mid i+2j = i'+2j' \wedge j' \le j-2\} \cup \\
&\quad \{[i,j] \to [i',j'] \mid \exists \beta \text{ s.t. } j+i' = i+j'+3\beta \wedge 6+i+j' \le j+i' \wedge 3+i+2j \le i'+2j'\} \cup \\
&\quad \{[i,j] \to [i', j-i+i'] \mid i \le i'-2\} \cup \\
&\quad \{[i,j] \to [i',j'] \mid 3+i+2j = i'+2j' \wedge j' \le j\} \cup \\
&\quad \{[i,j] \to [i',j'] \mid \exists \beta \text{ s.t. } j+i' = i+j'+3\beta \wedge 3+i+j' \le j+i' \wedge 6+i+2j \le i'+2j'\} \\
d - d^{2+} &= \{[i,j] \to [i+2,j-1]\} \cup \{[i,j] \to [i+1,j+1]\}
\end{aligned}$$

The dependence from the write of a(i+3,j) to the read of a(i,j) is found redundant.

Fig. 3. Example of determining dependences that must be explicitly synchronized

The relation $d^{+} \circ d$, which we denote d^{2+}, therefore contains all pairs of iterations that are linked by a chain of synchronizations of length **two** or more will therefore not have to be explicitly synchronized. So, the dependences that we do have to explicitly synchronize are $d - d^{2+}$. Note that this is equivalent to computing the transitive reduction of d. An example of the technique is presented in Figure 3, an example from [1].

In cases where more complex dependence relations cause the transitive closure calculation to be inexact, we can still produce useful results. We can safely subtract a lower bound on the 2+ closure from the dependences and still produce correct (but perhaps conservative) synchronization.

Our approach improves on related work in the following ways:

1. We use tuple relations as an abstraction for data dependences rather than the more traditional dependence distance representation. This allows us to handle non-constant dependences, which previous work is not able to do (see Figure 4).
2. Using dependence relations also allows us to use our algorithm for multidimensional loops without having to make special checks in boundary conditions. Related work builds an explicit graph of a subset of the iteration space, with each node representing an iteration of the loop body, and each edge representing a dependence [5]. Redundancy is found either through taking the transitive closure or reduction of this graph, or using algorithms that search a subgraph starting at the first iteration. In a one-dimensional loop, provided all dependence distances are constant, it is simple to find a small subgraph such that if a dependence is redundant in the subgraph, it is redundant throughout the iteration space.

   But in a multidimensional loop, the existence of negative inner dependence

```
doacross i = 1, n
    doacross j = 1, m
        A(i,j+2*i) = A(i,j) + Z(i,j)
        B(i,j) = B(i,j-4) + Y(i,j)
```

$$
\begin{aligned}
d_{11} &= \{[i,j] \rightarrow [i, 2i+j] \mid 1 \le i \le n \wedge 2i+j \le m \wedge 1 \le j\} \\
d_{22} &= \{[i,j] \rightarrow [i, j+4] \mid 1 \le i \le n \wedge 1 \le j < m\} \\
d &= d_{11} \cup d_{22} \\
d^{+} &= \{[i,j] \rightarrow [i,j'] \mid \exists \beta \text{ s.t. } j' = j + 4\beta \wedge 1 \le i \le n \wedge 1 \le j \le j' - 4 \wedge j' \le m\} \cup \\
&\quad \{[i,j] \rightarrow [i, 2i+j] \mid 1 \le i \le n \wedge 2i+j \le m \wedge 1 \le j\} \\
d^{2+} &= d^{+} \circ d \\
&= \{[i,j] \rightarrow [i,j'] \mid \exists \beta \text{ s.t. } j + 4\beta = 2i + j' \wedge 1 \le i \le n \wedge j' \le m \wedge 1 \le j \wedge 4 + 2i + j \le j'\} \cup \\
&\quad \{[i,j] \rightarrow [i, 4i+j] \mid 1 \le i \le n \wedge 4i+j \le m \wedge 1 \le j\} \cup \\
&\quad \{[i,j] \rightarrow [i,j'] \mid \exists \beta \text{ s.t. } j + 4\beta = j' \wedge 1 \le i \le n \wedge 1 \le j \le j' - 8 \wedge j' \le m\} \\
d - d^{2+} &= \{[i,j] \rightarrow [i, 2i+j] \mid 1 \le i \le 3, n \wedge 2i+j \le m \wedge 1 \le j\} \cup \\
&\quad \{[i,j] \rightarrow [i, 2i+j] \mid \exists \beta \text{ s.t. } 0 = 1 + i + 2\beta \wedge 5 \le i \le n \wedge 1 \le j \wedge 2i+j \le m\} \cup \\
&\quad \{[i,j] \rightarrow [i, j+4] \mid 2 \le i \le n \wedge 1 \le j \le m - 4\}
\end{aligned}
$$

We find that d_{11} does not need to be enforced when i > 3 and i is even (and thus 2i is a multiple of 4.)

Fig. 4. Example of non-constant dependence distances and partial redundancy

distances (such as (1,-2)) can result in *non-uniformly* redundant synchronizations [1]. A chain of synchronizations may exist within part of the iteration space, but at the edges of the iteration space, the chain may travel outside the bounds of the loops, and so intermediate iterations in the chain do not execute; thus it is difficult to find a small graph that finds all uniform redundancy. Figure 5 shows an example of finding an alternate path to handle the boundary cases. Methods that search a small graph, but which may miss some redundancy when nesting is greater than 2 have been developed[5].

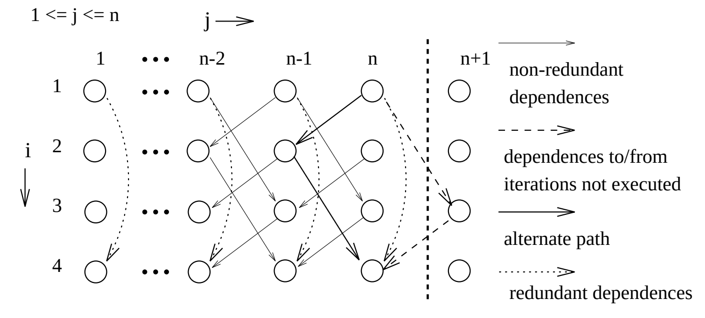
1 <= j <= n
j
1
n-2
n-1
n
n+1
1
2
3
4
i
non-redundant dependences
dependences to/from iterations not executed
alternate path
redundant dependences

Fig. 5. Finding alternate paths at boundaries; (3,0) is redundant when n > 1

Because we start with more precise dependence information, we do not have the same problem. No out-of-bounds iteration is in the range or domain of any dependence relation. Thus, we never need to worry that the 2+ closure will contain chains that are illegal at the edges of the iteration space. At the same time, since the 2+ closure contains all chains of two or more ordering constraints, all possible alternate paths are contained in it.

3. When a dependence is only partially redundant, we produce the conditions under which it needs to be explicitly enforced, and we can use that information to conditionally execute synchronization.

3.2 Testing the legality of iteration reordering transformations

Optimizing compilers reorder the iterations of statements so as to expose or increase parallelism and to improve data locality. An important part of this process is determining for each statement, which orderings of the iterations of that statement will preserve the semantics of the original code. Before we decide which orderings will be used for other statements, we can determine necessary conditions for the legality of an ordering for a particular statement by considering the direct self dependences of that statement. For example, it is not legal to interchange the i and j loops for statement 1 in Example 1 in Figure 6 because of the direct self dependence from $a(i-1, j+1)$ to $a(i, j)$. It is legal, however, to interchange the i and j loops for statement 2.

We can obtain stronger legality conditions by considering transitive self dependences, as is demonstrated by Example 2 in Figure 6. In this example, executing the i loop in reverse order is legal for both statements with respect to direct self dependences (there aren't any), but is not legal with respect to transitive self dependences.

To compute all transitive dependences we use an adapted form of the Floyd-Warshall algorithm for transitive closure. The algorithm is modified because we need to characterize each edge, not simply determine its existence. The algorithm is shown in Figure 7. In an iteration of the k loop, we update all dependences to incorporate all transitive dependences through statements $1..k$. The key expression in the algorithm is $d_{rq} \circ (d_{rr})^* \circ d_{pr}$. We include the $(d_{rr})^*$ term because we want to infer transitive dependences of the following form:

If there is a dependence from iteration i_1 of statement s_p to iteration

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

      Example 1
```

```
      do 2 i = 1, 4
1        a(i) = b(i)
2        b(i) = a(i-1)

      Example 2
```

Fig. 6. Examples of direct and transitive self dependences

```
for each statenment $r$
    for each statement $p$
        for each statement $q$
            $d_{pq} = d_{pq} \cup d_{rq} \circ (d_{rr})^* \circ d_{pr}$
```

Fig. 7. Modified Floyd-Warshall algorithm

i_2 of statement s_r and a chain of self dependences from iteration i_2 to iteration i_3 and finally a dependence from iteration i_3 to iteration i_4 of statement s_q then there is a transitive self dependence from iteration i_1 to iteration i_4.

3.3 General redundant synchronization removal

In this section, we consider a more general form of the problem described in Section 3.1. We no longer require the loops to be perfectly nested, the granularity of synchronization is now between iterations of particular statements (i.e. `posts` and `waits` occur immediately before and after the statements they are associated with) and we know how iterations will be distributed to the physical processors. For example, iterations may be distributed to a virtual processor array via a data distribution and the owner computes rule, and the virtual processor array may be folded onto the physical processor array in say a blocked fashion.

For each pair of statements p and q, we construct a relation that represents all ordering constraints on the iterations that are guaranteed to be satisfied in the distributed program. Such ordering constraints come from two sources:

1. If there is a data dependence from iteration i of statement p to iteration j of statement q (denoted $i \rightarrow j \in d_{pq}$), then i is guaranteed to be executed before j in any semantically equivalent distributed version of the program.
2. If iteration i of statement p and iteration j of statement q will be executed on the same physical processor (denoted $s_p(i) = s_q(j)$), and iteration i is executed before iteration j in the original execution order of the program (denoted $i \prec_{pq} j$), then i is guaranteed to be executed before j in the distributed program.

Combining these ordering constraints gives:

$$c_{pq} = d_{pq} \cup \{i \rightarrow j \mid i \prec_{pq} j \wedge s_p(i) = s_q(j)\}$$

Unlike in Section 3.1, we cannot determine which dependences need not be explicitly synchronized simply by computing $(c_{pq})^{2+}$. A synchronization may be redundant because of a chain of synchronizations through other statements. To determine such chains of ordering constraints, we first apply the algorithm in Figure 7 substituting c_{pq} for d_{pq} and producing c'_{pq}. This gives us all chains of ordering constraints of length **one** or more. We then find all chains of ordering constraints of length **two** or more using:

$$c''_{pq} = \bigcup_{r \in \{statements\}} c_{rq} \circ c'_{pr}$$

We do not need to explicitly synchronize iterations if they will be executed on the same physical processor, or if there is a chain of ordering constraints of length **two** or more. Therefore the only dependences that we have to synchronize explicitly are:

$$d_{pq} - \{ i \rightarrow j \mid s_p(i) = s_q(j) \} - c''_{pq}$$

If the number of physical processors is not known at compile time, the expression $s_p(i) = s_q(j)$ may not be affine. In such cases, we can instead use the stricter requirement that the two iterations will execute on the same virtual processor. This expression is always affine for the class of programs and distribution methods that we are able to handle and is a sufficient condition for the two iterations to be executed on the same physical processor. So, any redundancy that we find based on this stronger requirement can be safely eliminated.

Related work[6, 7, 1] considers the case of synchronization between statements with methods similar to the simple case. All of the methods build an explicit graph of a subset of the iteration space, with each node representing an iteration of a statement. Redundancy is found either by searching the graph[1] or using transitive closure of the graph[6, 7]; dependences are restricted to constant distances; and the problem regarding boundary cases still exists. These methods search a small graph which finds all redundancy when nesting level is 2, but may miss some redundancy when the nesting level is greater[1]. None of the above methods consider non-perfectly nested loops, and they do not use information regarding distribution. One previous technique has such the ability to generate the conditions under which a non-uniformly redundant dependence must be enforced[7], but the authors indicate that their technique may require taking transitive closure of a large subset of the iteration space.

3.4 Induction variables

Tuple relations and the transitive closure operation can also be used to compute closed form expressions for induction variables. We will use the program in Figure 8 as an example. In this example, we will be using 4-tuples because there are four scalar variables of interest in this program: i, j, n and m. For each edge in the control flow graph, we create a *state transition* relation which summarizes the change in value of the scalars as a result of executing the code in the control flow node corresponding to that edge and under what conditions execution occurs (see Figure 8). To investigate the state of the scalar variables at statement 6, we could use the algorithm in Figure 7 to compute (along with other things) all transitive edges from the start node to the node containing statement 6. Alternatively, we can directly calculate:

$$R_1 \bullet (R_2 \bullet (R_3 \bullet R4)^* \bullet R_5)^* \bullet R_2 \bullet (R_3 \bullet R_4)^* \bullet R_3$$

Which in this case evaluates to:

$$\{[i,j,p,q] \rightarrow [i',j',i'-1,20i'+2j'-20] \mid 2 \le i' \le n \wedge 1 \le j' \le 10\} \cup$$
$$\{[i,j,p,q] \rightarrow [1,j',n,2j'] \mid 1 \le j' \le 10 \wedge 1 \le n\}$$

```
1   q = 0
2   p = n
3   for i = 1 to n
4       for j = 1 to 10
5           q = q + 2
6           x[q] = y[p]
7       p = i
```

$R1 = \{[i,j,p,q] \rightarrow [1,j,n,0]\}$
$R2 = \{[i,j,p,q] \rightarrow [i,1,p,q] \mid i \leq n\}$
$R3 = \{[i,j,p,q] \rightarrow [i,j,p,q+2] \mid j \leq 10\}$
$R4 = \{[i,j,p,q] \rightarrow [i,j+1,p,q]\}$
$R5 = \{[i,j,p,q] \rightarrow [i+1,j,i,q] \mid j > 10\}$
$R6 = \{[i,j,p,q] \rightarrow [i,j,p,q]\ i > n\}$

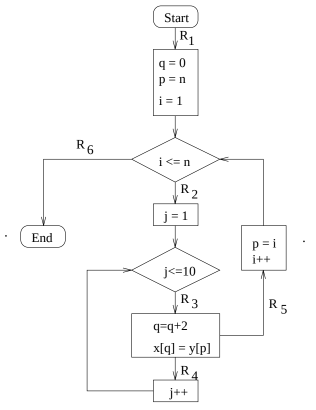

Fig. 8. Induction Variable Example

From this result, we can deduce that at line 6 we can replace the induction variable p with `(i=1?n:i-1)` and the induction variable q with `20i+2j-20`.

This general approach has uses other that induction variable recognition, such as deriving or proving assertions about scalar variables. The fact that we could use transitive closure to potentially completely describe the effect of arbitrary programs consisting of loops and conditionals with affine bounds and conditions and assignment statements involving affine expressions further demonstrates that transitive closure cannot always be computed exactly, since such analysis is known to be uncomputable.

4 Computing the Transitive Closure of a Single Relation

In this section we describe techniques for computing the positive transitive closure of a relation. The transitive closure R^* can be computed from the positive transitive closure R^+ as $R^+ \cup I$, where I is the identity relation. In the following text we will use the term transitive closure for both R^+ and R^*. The difference will be evident from the context.

The exact transitive closure R^+ of a relation R can be equivalently defined as $R^+ = \bigcup_{k=1}^{\infty} R^k$, where $R^k = \underbrace{R \circ R \circ \ldots \circ R}_{k\ times}$. We will shortly describe techniques that will often compute R^+ exactly. In situations where they do not apply, we can produce increasingly accurate lower bounds using the following formula:

$$R^+_{LB(n)} = \bigcup_{k=1}^{n} R^k \qquad (1)$$

In some cases $R^+_{LB(n)} = R^+$ for all n greater than some small value. The following theorem allows us to determine when a lower bound is equal to the exact transitive closure:

Theorem 1. *For all relations P and R such that $R \subseteq P \subseteq R^+$ the following holds: $P = R^+$ if and only if $P \circ R \subseteq P$.*

Proof. The "only if" part is trivial. To prove the "if" part we will prove by induction on k that $R^k \subseteq P$. The assumption $R \subseteq P$ proves the base case. If $R^k \subseteq P$ then $R^{k+1} = (R^k \circ R) \subseteq (P \circ R) \subseteq P$. Since $R^+ = \bigcup_{k=1}^{\infty} R^k$ and $\forall k \geq 1, R^k \subseteq P$, we know that $R^+ \subseteq P$. Thus $P = R^+$. □

Corollary 2. $R^+_{LB(n)} = R^+$ *iff* $R^+_{LB(n)} \circ R \subseteq R^+_{LB(n)}$.

Thus, one approach to computing transitive closure would be to compute more and more accurate lower bounds until the result becomes exact. Although this technique works in some cases, there is no guarantee of termination. For example, the exact transitive closure of $R = \{[i] \rightarrow [i+1]\}$ cannot be computed using this approach. Thus more sophisticated techniques are required. Section 4.1 describes techniques that work in the special case of relations that can be described by a single conjunct. Section 4.2 describes techniques for the general case, making use of the techniques used for the single conjunct case.

4.1 Single conjunct relations

For a certain class of single conjunct relations, the transitive closure can be calculated straightforwardly. Consider the following example:

$$R = \{[i_1, i_2] \rightarrow [j_1, j_2] \mid j_1 - i_1 \geq 2 \wedge j_2 - i_2 = 2 \wedge \exists \alpha \text{ s.t. } j_1 - i_1 = 2\alpha\}$$

For any $k \geq 1$ the relation R^k can be calculated as:

$$R^k = \{[i_1, i_2] \rightarrow [j_1, j_2] \mid j_1 - i_1 \geq 2k \wedge j_2 - i_2 = 2k \wedge \exists \alpha \text{ s.t. } j_1 - i_1 = 2\alpha\}$$

By making k in the above expression existentially quantified, we get the union of R^k for all $k > 0$; that is, R^+:

$$\{[i_1, i_2] \rightarrow [j_1, j_2] \mid \exists k > 0 \text{ s.t. } j_1 - i_1 \geq 2k \wedge j_2 - i_2 = 2k \wedge \exists \alpha \text{ s.t. } j_1 - i_1 = 2\alpha\}$$

This method can be used for any relation that only contains constraints on the differences between the corresponding elements of the input and output tuples. We call such relations *d-form relations.*

Definition 3. A relation R is said to be in d-form iff it can be written as:

$$\{[i_1, i_2, \ldots i_m] \rightarrow [j_1, j_2 \ldots j_m] \mid \forall p, 1 \leq p \leq m,\ L_p \leq j_p - i_p \leq U_p \wedge j_p - i_p = M_p\ \alpha_p\ \}$$

where L_p and U_p are constants and M_p is an integer. If L_p is $-\infty$ or U_p is $+\infty$ the corresponding constraints are not included in the above equation.

The transitive closure of a d-form relation is:

$$\{[i_1, i_2, \ldots i_m] \rightarrow [j_1, j_2 \ldots i_m] \mid \exists k > 0 \text{ s.t. } \forall p, 1 \leq p \leq m, L_p k \leq j_p - i_p \leq U_p k \wedge j_p - i_p = M_p\ \alpha_p\ \} \quad (2)$$

For any relation R, it is always possible to compute a d-form relation d such that $R \subseteq d$. We can then use d^+ as an upper bound on R^+ since for any two

$$\begin{array}{ll}
R & = \{[i_1,i_2] \rightarrow [j_1,j_2] \mid j_1 - i_1 = 1 \wedge j_2 - i_2 \geq 2 \wedge 1 \leq i_1, j_1, j_2 \leq n \wedge \\
 & \quad i_1 \leq i_2 \leq n\} \\
d & = \{[i_1,i_2] \rightarrow [j_1,j_1] \mid j_1 - i_1 = 1 \wedge j_2 - i_2 \geq 2\} \\
d^+ & = \{[i_1,i_2] \rightarrow [j_1,j_2] \mid i_1 < j_1 \wedge j_2 - i_2 \geq 2(j_1 - i_1)\} \\
Domain(R) & = \{[i_1,i_2] \mid 1 \leq i_1 \leq i_2 \leq n-2\} \\
Range(R) & = \{[j_1,j_2] \mid 2 \leq j_1 < j_2 \leq n\} \\
h & = \{[i_1,i_2] \rightarrow [j_1,j_2] \mid 1 \leq i_1 \leq i_2 \leq n-2 \wedge 2 \leq j_1 < j_2 \leq n\} \\
D_+ & = \{[i_1,i_2] \rightarrow [j_1,j_2] \mid 1 \leq i_1 \leq i_2 \wedge i_1 < j_1 \wedge j_2 \leq n \wedge \\
 & \quad j_2 - i_2 \geq 2(j_1 - i_1)\}
\end{array}$$

D_+ is lexicographically forward and $D_+ \subseteq R \circ D_+ \cup R$, thus $R^+ = D_+$.

Fig. 9. Example of calculating transitive closure of a single conjunct relation

relations R_1 and R_2, if $R_1 \subseteq R_2$ then $R_1^+ \subseteq R_2^+$. To improve this upper bound we can restrict the domain and range of d^+ to those of R by computing $D_+ = d^+ \cap h$, where $h = Domain(R) \times Range(R)$.

In most of our applications the relations R and D_+ have the property of being *lexicographically forward.*

Definition 4. A relation A is lexicographically forward iff $\forall x \rightarrow y \in A, 0 \prec y - x$ ($y - x$ is lexicographically positive).

For lexicographically forward relations we can check whether the upper bound is an exact transitive closure using the following theorem:

Theorem 5. $\forall$ *lexicographically forward relations* P, R : $\quad R \subseteq P \Rightarrow (P = R^+ \Leftrightarrow P \subseteq (R \cup R \circ P))$.

Proof. Firstly, we prove that $(P \subseteq (R \cup R \circ P)) \Rightarrow (P \subseteq R^+)$. Rewriting this proposition in the terms of the relation elements yields: $(\forall x \rightarrow z \in P, x \rightarrow z \in R \vee \exists y \text{ s.t. } x \rightarrow y \in R \wedge y \rightarrow z \in P) \Rightarrow (\forall x \rightarrow z \in P x \rightarrow z \in R^+)$. This is proved by induction on $z - x$ using lexicographical ordering. The base case $x = z$ is vacuous. For the induction step consider any $x \rightarrow y \in P$. If $x \rightarrow z \in R$ then $x \rightarrow z \in R^+$. Otherwise, $\exists y$ s.t. $x \rightarrow y \in R \wedge y \rightarrow z \in P$. Because P and R are lexicographically forward, $z - x \succ z - y$. Thus, by induction hypothesis, $y \rightarrow z \in R^+$, and, consequently $x \rightarrow z \in R^+$, i.e. $P \subseteq R^+$. $R^+ \subseteq P$ is in the theorem assumption, so $P = R^+$. The reverse statement $P = R^+ \Rightarrow P \subseteq R \circ P \cup R$ is trivial. □

Corollary 6. *If* D_+ *is lexicographically forward, then*

$$D_+ = R^+ \text{ iff } D_+ \subseteq (R \cup R \circ D_+) \tag{3}$$

4.2 Multiple conjunct relations

Computing the transitive closure of a relation with more than one conjunct via a naive application of Equation 1 is prohibitively expensive due to the possible exponential growth in number of the conjuncts. We have developed techniques that try to limit this growth. We first describe how to compute the transitive closure of a two conjunct relation; then we show how to generalize this technique for relations with an arbitrary number of conjuncts.

```
Input: $R = \bigcup_{i=1}^{m} C_i$
Output: $R^+$ or $R^+_{LB}$
Invariant: $(R^+ \supseteq T \cup W^+) \wedge (exact \Rightarrow R^+ = T \cup W^+)$
$T = \emptyset$; $W = R$; exact= $true$
while not ($W = \emptyset$ or "accept $W$ as $W^+_{LB}$") do
    choose a conjunct $A \in W$; remove $A$ from $W$
    if $A^+$ is known then
        $T = T \cup A^+$
        $W_{new} = \emptyset$
        for all conjuncts $C_i \in W$ do
            if $(A^? - A^+) \circ C_i \circ (A^? - A^+) \equiv C_i$ then $W_{new} = W_{new} \cup (A^? \circ C_i \circ A^?)$
            else if $C_i \circ (A^? - A^+) \equiv C_i$ then $W_{new} = W_{new} \cup (C_i \circ A^?) \cup (A^+ \circ C_i \circ A^?)$
            else if $(A^? - A^+) \circ C_i \equiv C_i$ then $W_{new} = W_{new} \cup (A^? \circ C_i) \cup (A^? \circ C_i \circ A^+)$
            else $W_{new} = W_{new} \cup (C_i \circ A^+) \cup (A^+ \circ C_i) \cup (A^+ \circ C_i \circ A^+) \cup C_i$
        endfor
        $W = W_{new}$
    else
        $T = T \cup A^+_{LB}$
        $W = (W \circ A^+_{LB}) \cup (A^+_{LB} \circ W \cup A^+_{LB}) \circ (W \circ A^+_{LB}) \cup W$
        exact = $false$
endwhile
if ($W = \emptyset$ and exact = $true$) or $(T \cup W) \circ (T \cup W) \subseteq (T \cup W)$ then
    $R^+ = T \cup W$
else
    $R^+_{LB} = T \cup W$
```

Fig. 10. The algorithm for computing transitive closure

Computing the transitive closure of two-conjunct relations Let R be a two-conjunct relation, $R = C_1 \cup C_2$. The transitive closure of R is:

$$(C_1 \cup C_2)^+ = C_1^+ \cup (C_1^* \circ C_2 \circ C_1^*)^+ \tag{4}$$

If $C_1^* \circ C_2 \circ C_1^*$ is a single conjunct relation, its closure can be calculated using the techniques described in the Section 4.1. Unfortunately, C_1^* is often not a single conjunct relation even if C_1^+ is. To overcome this difficulty, we use a single conjunct approximation of C_1^*, that we will denote $C_1^?$ and call ?-*closure*. We try to select an $C_1^?$ that has the following desirable property

$$C_1^* \circ C_2 \circ C_1^* \equiv C_1^? \circ C_2 \circ C_1^? \tag{5}$$

If this is the case, we can use $C_1^?$ instead of C^* in Equation 4. If not, it may still be possible to limit the number of conjuncts in $(C_1 \cup C_2)^+$ through the use of $C_1^?$ if $C_1^* \circ C_2 \equiv C_1^? \circ C_2$ or $C_2 \circ C_1^* \equiv C_2 \circ C_1^?$. Testing the property described in Equation 5 directly is rather expensive, so instead we test a weaker predicate: $(C_1^? - C_1^+) \circ C_2 \circ (C_1^? - C_1^+) \equiv C_2$.

Heuristics for computing ?-closure We try to compute ?-closure for a relation R only if R^+ is a single conjunct relation. In such cases we are trying to compute a $R^?$ that is a superset of R^+ and includes some elements from I and also some other elements required to make it a single conjunct relation. In many cases, these additional elements do not affect the result of the composition.

$$\begin{array}{ll}
R & = \{[i,j] \to [i',j+1] \mid 1 \le i,j,j+1 \le n \wedge i' = i\} \cup \\
 & \quad \{[i,n] \to [i+1,1] \mid 1 \le i, i+1 \le n\} \\
C_1^+ & = \{[i,j] \to [i,j'] \mid 1 \le j < j' \le n \wedge 1 \le i \le n\} \\
C_1^? & = \{[i,j] \to [i,j'] \mid 1 \le j \le j' \le n \wedge 1 \le i \le n\} \\
C_1^? \circ C_2 \circ C_1^? & = \{[i,j] \to [i+1,j']' \mid 1 \le i < n \wedge 1 \le j \le n \wedge 1 \le j' \le n\} \\
\text{Since,} & \quad C_1^? \circ C_2 \circ C_1^? \equiv C_1^* \circ C_2 \circ C_1^* \\
(C_1^* \circ C_2 \circ C_1^*)^+ & = (C_1^? \circ C_2 \circ C_1^?)^+ \\
 & = \{[i,j] \to [i',j'] \mid 1 \le i < i' \le n \wedge 1 \le j, j` \le n\} \\
R^+ & = C_1^+ \cup (C_1^* \circ C_2 \circ C_1^*)^+ \\
 & = \{[i,j] \to [i',j'] \mid (1 \le j < j' \le n \wedge 1 \le i \le n \wedge i = i')\} \cup \\
 & \quad \{[i,j] \to [i',j'] \mid (1 \le i < i' \le n \wedge 1 \le j \le n \wedge 1 \le j' \le n)\}
\end{array}$$

Fig. 11. Example of transitive closure calculation

For a d-form relation d (see Section 4.1) we compute $d^?$ as:

$$\{[i_1, i_2, \ldots i_m] \to [j_1, j_2 \ldots i_m] \mid \exists k \ge 0 \text{ s.t. } \forall p, 1 \le p \le m, L_p k \le j_p - i_p \le U_p k \wedge j_p - i_p = M_p \alpha_p)\} \quad (6)$$

For a relation R s.t. $R^+ = D_+$, we calculate $R^?$ by restricting the domain and range of $d^?$ to a single conjunct tuple set h' that contains both the domain and range of R. In other cases we assume that $? - closure$ cannot be computed and set $R^?$ to $\emptyset$.

Computing the transitive closure of multiple conjunct relations The transitive closure of a relation with an arbitrary number of conjuncts can be computed similarly to the transitive closure of a relation with two conjuncts. Let R be a m-conjunct relation $R = \bigcup_{i=1}^{m} C_i$. Its transitive closure is:

$$R^+ = C_1^+ \cup (C_1^* \circ \bigcup_{i=2}^{m} C_i \circ C_1^*)^+ = C_1^+ \cup (\bigcup_{i=2}^{m} C_1^* \circ C_i \circ C_1^*)^+$$

For $i \in \{2, \ldots, m\}$, $C_1^* \circ C_i \circ C_1^*$ can be computed using the techniques described in the two conjunct case. After all these terms are computed, the same algorithm can be applied recursively to compute the transitive closure of their union. The algorithm is shown in Figure 10. The algorithm will terminate when the transitive closure has been computed exactly or when we are willing to accept the current approximation as a lower bound. In many cases, what we accept as a lower bound turns out to be exact after all, and can be proved to be so using Theorem 1. An example of a transitive closure calculation using this algorithm is shown in Figure 11. The order in which we consider the conjuncts in a relation can significantly affect the performance of our algorithm. One heuristics that we use is to consider first those conjuncts C_i for which we can find a $C_i^?$ that satisfies the Equation 5. In some cases, pre-computing positive transitive closure of some of the conjuncts in the original relation can also simplify the calculations.

The above algorithm allows us to compute the exact transitive closure of a multiple conjunct relation or its lower bound. If an upper bound is required, it can be calculated in a manner similar to that of the single conjunct relation.

5 Conclusion

We have presented a number of applications for the transitive closure of tuple relations. These applications include:

- Avoiding redundant synchronization of iterations executing on different processors.
- Precisely describing which iterations of a statement are data dependent on which other iterations, and using this information to determine which iteration reordering transformations are legal.
- Computing closed form expressions for induction variables.

We also presented algorithms for transitive closure that produce exact results in most commonly occurring cases and produce upper or lower bounds (as necessary) in the other cases. Our preliminary experiments show that we produce exact results for most of the programs we have considered. We will provide detailed experiential results in the final version of this paper. We believe that the applications described in this paper are only a small subset of the possible applications of this general purpose program abstraction and set of operations.

References

1. Ding-Kai Chen. *Compiler Optimizations for Parallel Loops With Fine-Grained Synchronization.* PhD thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, 1994. Also available as CSRD Report 1374.
2. Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The omega library interface guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, March 1995.
3. 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.
4. Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *Lecture Notes in Computer Science 892: Seventh International Workshop on Languages and Compilers for Parallel Computing,* Ithaca, NY, August 1994. Springer-Verlag.
5. V.P. Krothapalli and P. Sadayappan. Removal of redundant dependences in DOACROSS loops with constant dependences. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pages 51–60, July 1991.
6. S.P. Midkiff and D.A. Padua. Compiler algorithm for synchronization. *IEEE Trans. on Computers*, C-36(12):1485–1495, 1987.
7. S.P. Midkiff and D.A. Padua. A comparison of four synchronization optimization techniques. In *Proc. 1991 IEEE International Conf. on Parallel Processing,* pages II–9 – II–16, August 1991.

This article was processed using the LaTeX macro package with LLNCS style