International Journal of Parallel Programming, Vol. 24, No. 6, 1996

Transitive Closure of Infinite Graphs and Its Applications

Wayne Kelly,[1] William Pugh,[1] Evan Rosser,[1] and Tatiana Shpeisman[1]

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 relation $\{[i] \rightarrow [i+2] | 1 \leqslant i \leqslant n-2\}$. Even when it is possible to enumerate the related pairs of tuples, such as for the relation $\{[i, j] \rightarrow [i', j'] | 1 \leqslant i, j, i', j' \leqslant 100\}$, 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 $\{[i] \rightarrow [i'] | \exists \beta \text{ s.t. } i' - i = 2\beta \wedge 1 \leqslant i \leqslant i' \leqslant n\}$. As we will prove, transitive closure is not comutable in the general case. We have developed algorithms that produce exact results in most commonly occuring cases and produce upper of lower bounds (as necessary) in the other cases. This paper will describe ou algorithms for computing transitive closure and some of its applications such as determining which inter-processor synchronizations are redundant.

KEY WORDS: Integer tuple relations; transitive closure; dependence relations; redundant sychronization removal.

[1] Department of Computer Science, University of Maryland, College Park, Maryland 20742. E-mail: {wak,pugh,ejr,murka}@cs.umd.edu.

0885-7458/96/1200-0579$09.50/0

1. INTRODUCTION

This paper proposes a new general purpose abstraction called tuple relations, that is capable of concisely summarizing many kinds of information gathered from analysis of scientific codes. We provide a number of operations on these tuple relations including a particularly powerful one called transitive closure. We will show how transitive closure leads to simple and elegant solutions to several program analysis problems.

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:

$$\{[i] \rightarrow [i', j'] | 1 \leqslant i = i' = j' \leqslant n\}$$

One possible use of tuple relations is to concisely and accurately represent the data dependences in a program. For example, the relation given here describes the data dependences from statement 1 to statement 2 in the program shown in Fig. 1.

We use the term *dependence relation* rather than tuple relation when relations describe data dependences. A dependence relation is a much more powerful abstraction than 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.[1]

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 taking the transitive closure of these relations is a key element in removing redundant synchronizations within loops. In the case of perfectly nested loops with the entire loop body considered as an atomic statement, doing so is not prohibitively expensive and gives us more power than any of the previous papers on this topic. We also describe exact methods for

```
        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.

removing redundant synchronization for the more general case of imperfectly nested loops with synchronization occurring between individual statements. But just because our methods might be capable of computing this exactly doesn't mean that they can or should only be used in this way. It is a relatively straightforward process to extend our exact method into a framework in which accuracy can be traded off for efficiency.

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 deals with 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:

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

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

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

Table I 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.edu/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:

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

Table I. Operations on Tuple Relations

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

and *positive transitive closure* operator:

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

In previous work,[3] 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:

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

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.

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

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:

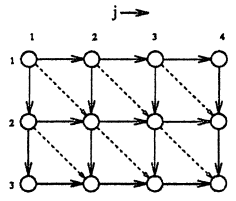

Fig. 2. Example of redundant synchronization.

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 orginal 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 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[4] 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 Fig. 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

```
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)
```

$$d = \{[i,j] \rightarrow [i+3,j]\} \cup \{[i,j] \rightarrow [i+2,j-1]\} \cup \{[i,j] \rightarrow [i+1,j+1]\}$$

$$d^{2+} = \{[i,j] \rightarrow [i',j'] \mid \exists a,b,c \text{ s.t. } i' = 3a+2b+c+i \wedge j' = -b+c+j \wedge a \geq 0 \wedge b \geq 0 \wedge c \geq 0 \wedge a+b+c > 1\}$$

$$= \{[i,j] \rightarrow [i',j'] \mid \exists \alpha \text{ s.t. } i+j' = j+i'+3\alpha \wedge i+2j \leq i'+2j' \wedge 6+2i+j \wedge 2i'+j' \wedge i+j' \leq j+i')\}$$

$$d - d^{2+} = \{[i,j] \rightarrow [i+2,j-1]\} \cup \{[i,j] \rightarrow [i+1,j+1]\}$$

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.

length one or more. 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 and 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 Fig. 3, an example from Ref. 5. In simple cases such as in Fig. 3, we can compute the $2+$ closure directly (without using the general purpose techniques in Section 4), to get a simpler form for the relation. Intuitively, a path in d^+ is made up of individual arcs from d, each of which is attributable to one of the conjuncts in d. To compute d^+ directly, we introduce an existentially quantified variable for each conjunct c in d, where the new variable represents the number of arcs followed from c for a path in d^+. Each of these variables is constrained to be ≥ 0 (there are no backward arcs) and their sum is constrained to be >0 (at least one arc from one conjunct in each path). For d^{2+}, we constrain their sum to be >1, so at least two arcs must be followed for a path to be included. The variables a, b, and c serve this purpose in Fig. 3.

In cases where complex dependence relations cause the transitive closure calculation to be inexact, we can still produce useful results. We can safely substract 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 Fig. 4).

2. 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. Dependence relations contain all conditions which must be satisfied in order for the dependence to exist, including those concerning the existence of the dependence at the edges of the iteration space. Thus, we can apply the algorithm to nested loops without having to make special checks in boundary cases.

A further discussion of related methods using transitive closure of finite graphs helps explain the third point. These methods build an explicit graph of a subset of the iteration space; each node represents an iteration of the loop body, and each edge represents a dependence.[4] 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 nested loop, the existence of negative inner dependence distances (such as $(1, -2)$) can result in *nonuniformly* redundant synchronizations.[5] A chain of synchronizations may exist within part of the iteration space, but at the edges of the iteration space, that chain may travel outside the bounds of the loops, and so intermediate iterations in the chain do not

```
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)
```

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

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 nonconstant dependence distances and partial redundancy.

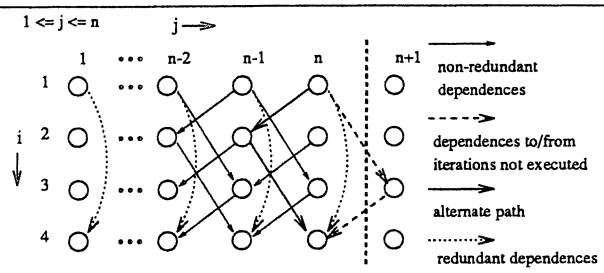

Fig. 5. Finding alternate paths at boundaries; (3,0) is redundant when $n > 1$.

execute; thus it is difficult to construct 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.[4]

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, we consider all possible alternate paths.

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

```
    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
        a(i) = b(i)                1
        b(i) = a(i-1)              2
```

Example 2

Fig. 6. Examples of direct and transitive self dependences.

```
for each statement r
    for each statement p
        for each statement q
            d_pq = d_pq ∪ d_rq ∘ (d_rr)* ∘ d_pr
```

Fig. 7. Modified Floyd-Warshall algorithm.

statement. For example, it is not legal to interchange the i and j loops for statement 1 in Example 1 in Fig. 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 Fig. 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. We denote by d_{pq} the data dependences from statement s_p to statement s_q. The algorithm is shown in Fig. 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 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.

In Ref. 6 we describe a framework for unifying iteration reordering transformations that uses a legality test similar to those described earlier.

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, we might know that iterations are distributed to a virtual processor array via a data distribution

and the owner computes rule, and the virtual processor array is folded onto the physical processor array in a blocked fashion.

For each pair of statements p and q, we construct a relation c_{pq} 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 subtituting 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[5, 7, 8] addresses the case of synchronization between statements using methods similar to those used for the simple case. Most 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[5] or using transitive closure of the graph[7, 8]; 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.[5] None of these methods consider imperfectly nested loops, and they do not use information regarding distribution. One previous technique has the ability to generate the conditions under which a nonuniformly redundant dependence must be enforced[8], but the authors indicate that their technique may require taking transitive closure of a large subset of the iteration space.

Another method removes synchronization for a regular pattern of communication between processors when using put communication instead of send and receive, and in some cases can remove all synchronization even when communication exists.[9] Our framework can be extended to incorporate the information that all flow dependences will be implicitly synchronized by a put, and use that to eliminate other types of dependences. We can also incorporate information about barrier synchronization in our relations to remove redundant point-to-point synchronization in their presence.[10]

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 Fig. 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, p, and q. 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 the conditions under which execution occurs (see Fig. 8). To investigate the state of the scalar variables at statement 6, we could use the algorithm in Fig. 7 to compute (along with other things) all transitive edges from the

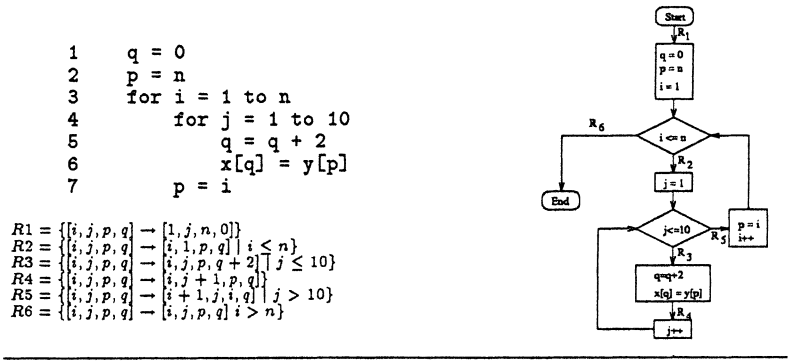

Fig. 8. Induction variable example.

start node to the node containing statement 6. Alternatively, we can directly calculate:

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

Which in this case evaluates to:

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

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

This general approach has uses other than 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 $R^k = \underbrace{R \circ R \circ \ldots \circ R}_{k \text{ times}}$. 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:

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

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 $\forall k \geqslant 1$, $R^k \subseteq P$, 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 $R = \{[i] \to [i+1]\}$ 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

4.1.1. Computing the Upper Bound on the Transitive Closure

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

$$R = \{[i_1, i_2] \to [j_1, j_2] | j_1 - i_1 \geqslant 2 \wedge j_2 - i_2 = 2 \wedge \exists \alpha \text{ s.t. } j_1 - i_1 = 2\alpha\}$$

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

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

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

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

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:

$$\{[i_1, i_2, \ldots, i_m] \rightarrow [j_1, j_2, \ldots, j_m] | \forall p, a \leqslant p \leqslant m, \\ L_p \leqslant j_p - i_p \leqslant U_p \wedge \exists \alpha_p \text{ s.t. } j_p - i_p = M_p \alpha_p\}$$

where L_p is an integer or $-\infty$, U_p is an integer or $+\infty$ and M_p is an integer.

The transitive closure of a d-form relation is:

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

For any relation R that is not in d-form, it is relatively easy to compute a d-form relation d such that $R \subseteq d$. For each $p \in \{1, \ldots, m\}$, we introduce a new variable equal to $j_p - i_p$ and project away all other variables. We then look for upper and lower bounds and stride constraints on this variable. So, it is always possible to compute a d-form relation that is a superset of R, since in the worst case we can set M_p to 1, L_p to $-\infty$ and U_p to $+\infty$. We can then use d^+ as an upper bound on R^+ since for any two 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 $h = Domain(R) \times Range(R)$.

4.1.2 Checking Whether the Upper Bound is an Exact Transitive Closure

We want to be able to check to see if D_+ (an upper bound) is an exact transitive closure.

Lemma 4. If $P = P^+$ and $P \cap I = \varnothing$ then P is acyclic.

Proof. If there is a path from x to x in P, then since P is transitively closed, $x \rightarrow x \in P$.

Theorem 5. For any relations R and P such that P is acyclic and $R \subseteq P \subseteq R \cup P \circ R$, we prove $P \subseteq R^+$.

Proof. For any $x \to z \in P$, we'll prove $x \to z \in R^+$. We know $x \to z \in R \cup P \circ R$.

The inductive proof is based on the length of the longest path from x to z through P. Because P is acyclic this is a finite number for each given x and z. Since $x \to z \in P$, there exists at least one path of length 1.

Base case: When $x \to z$ has a maximum path length of 1, $x \to z \notin P \circ P$. Since $R \subseteq P$, we know $P \circ R \subseteq P \circ P$ and therefore $x \to z \notin P \circ R$. Therefore, $x \to z \in R \subseteq R^+$.

Inductive case: ($x \to z$ has a maximum path length greater than 1) We consider two cases:

1. If $x \to z \in R$, then $x \to z \in R^+$.
2. If $x \to z \in P \circ R$, then there exists a y such that $x \to y \in R \subseteq P$ and $y \to z \in P$. The maximum length from x to z in P must be at least one more than the maximum path length from y to z in P. Therefore, by our inductive hypothesis, $y \to z \in R^+$ and therefore $x \to z \in R^+$. □

Theorem 6. For any relations R and P such that P is acyclic and $R^+ \subseteq P$, we prove $R^+ = P$ if and only if $P \subseteq R \cup P \circ R$.

Proof. The proof that $P = R^+ \Rightarrow P \subseteq (R \cup P \circ R)$ is trivial. We need to prove that $P \subseteq R \cup P \circ R$ implies $R^+ = P$. Since $R \subseteq R^+ \subseteq P$, we satisfy the conditions of Theorem 5 and we derive that $P \subseteq R^+$. Since we also know that $R^+ \subseteq P$, we prove $R^+ = P$.

Corollary 7. If $D_+ \cap I = \varnothing$, then

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

We collected statistics for all of the examples given in this paper, plus about 2000 other real-life examples of transitive closure that arose from our applications (primarily dependence analysis). This test showed that 97% of single conjunct closures performed in these examples were computed exactly.

4.2. Multiple Conjunct Relations

Computing a lower bound on the transitive closure of a relation with more than one conjunct via a naive application of Eq. 1 is prohibitively expensive due to the possible exponential growth in the 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.

4.2.1. 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:

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

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. $C_1^* \circ C_2 \circ C_1^*$ will be a single conjunct relation provided that C_1^* is a single conjunct relation, since the composition of two single conjunct relations is always a single conjunct relation. 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 a $C_1^?$ that has the following desirable property

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

Our choice of $C_1^?$ will not depend on C_2, so there is only hope, not a guarantee, of satisfying this property. If this hope is realized then we can use $C_1^?$ instead of C_1^* in Eq. 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 Eq. 5 directly is rather expensive, so we instead use the following cheaper but possibly conservative test. If $(C_1^? - C_1^+)$ is convex (which is often the case) and $(C_1^? - C_1^+) \circ C_2 \circ (C_1^? - C_1^+) \equiv C_2$ then we know that the property described in Eq. 5 holds; otherwise we assume it doesn't. This test succeeded in all of the 2000-plus examples described earlier.

4.2.2. Heuristics for Computing ?-Closure

We try to compute ?-closure for a relation R only if R^+ is an exact single conjunct relation. We do it by using a d-form relation (see Section 4.1).

For a d-form relation d we can compute d^* as:

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

For a relation R s.t. $R^+ = D_+$, we use $R^? = d^* \cap h'$, where $h' = (Domain(R) \cup Range(R)) \times (Domain(R) \cup Range(R))$. Thus

$$R^? = R^+ \cup (I \cap h') \cup (d^+ \cap (h' - h)) \tag{7}$$

Although under certain conditions $R^? = R^*$, in general this is not the case. Note, that any relation R^* includes identity while $R^?$ includes just some elements of it unless h' is a total set. Still, property 5 will often hold, since the missing elements from I are often not in the domain or range of C_2. The additional elements in $R^?$ (third term of 7) may not exist or may not affect the result of the composition.

If a relation R is s.t. $R^+ \neq D_+$ we assume that *?-closure* cannot be computed and let $R^?$ be the empty relation.

4.2.3. 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 an m-conjunct relation $R = \bigcup_{i=1}^{m} C_i$. Its transitive closure is:

$$R^+ = C_1^+ \cup \left(C_1^* \circ \bigcup_{i=2}^{m} C_i \circ C_1^* \right)^+ = C_1^+ \cup \left(\bigcup_{i=2}^{m} C_1^* \circ C_i \circ C_1^* \right)^+ \qquad (8)$$

For $i \in \{2,..., m\}$, $C_1^* \circ C_i \circ C_1^*$ 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 Fig. 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. We computed the exact transitive closure in 99% of the examples described earlier. An example of a transitive calculation using this algorithm is shown in Fig. 12.

The order in which we consider the conjuncts in a relation can significantly affect the performance of our algorithm. One heuristic that we use is to consider first those conjuncts C_i for which we can find a $C_i^?$ that satisfies Eq. 5. In some cases, pre-computing the positive transitive closure

---

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

$D_+ \cap I$ is empty and $D_+ \subseteq R \circ D_+ \cup R$, thus $R^+ = D_+$.

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

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
  //Set $T, W = T \cup A^+, A^* \circ W \circ A^*$
  if we can determine A^+ exactly **then**
    $T = T \cup A^+$
    $W_{new} = \emptyset$
    for all conjuncts $C_i \in W$ **do**
      // $W_{new} = W_{new} \cup A^* \circ C_i \circ A^*$
      // See Section 4.2
      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
// Use Corollary 2 to see if exact
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.

of some of the conjuncts in the original relation (i.e., replacing C_i by C_i^+) can also simplify the calculations.

The algorithm in Fig. 10 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 a single conjunct relation.

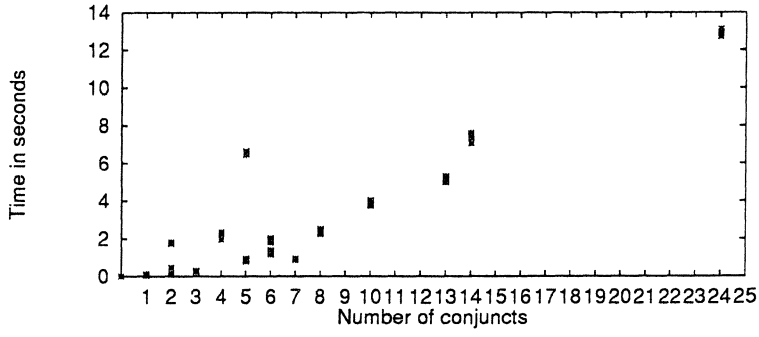

Fig. 11. Relationship between number of conjuncts and computation time.

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

Fig. 12. Example of transitive closure calculation.

The time required to compute transitive closure obviously increases with the number of conjuncts in the original. Figure 11 shows the time taken by our algorithm, as implemented in C++ as part of the Omega Library, for the entire set of examples described earlier. Experiments were performed on a Sparc 10/51 with 64MB of main memory.

5. CONCLUSIONS

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

- 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.

While problems such as induction variable recognition and redundant synchronization elimination might be better handled in practice by other more specific methods, we include them here to demonstrate that in both cases, the fundamental problem to be solved is transitive closure. Once we have discovered this, we can develop specific algorithms, or apply transitive closure in a controlled way.

We have also presented algorithms for transitive closure that produce exact results in most commonly occuring cases and produce upper or lower bounds (as necessary) in the other cases. Our experiments show that we produce exact results for most of the programs we have considered. Future work is needed to ensure that we can efficiently generate concise results in more cases.

Tuple relations are a useful, general purpose program abstraction, and transitive closure is a particularly powerful operation that can lend insight into the nature of these problems, as well as be a useful tool for their solution. We believe that the applications described here are only a small subset of what is possible.

6. AVAILABILITY

The transitive closure algorithms are implemented in the Omega Library, which is available from ftp://ftp.cs.umd.edu/pub/omega.

REFERENCES

1. Wayne Kelly, and William Pugh, A Framework for Unifying Reordering Transformations. Technical Report CS-TR-3193, Department of Computer Science, University of Maryland, College Park (April 1993).
2. Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott, The Omega Library Interface Guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park (March 1995).
3. Wayne Kelly and William Pugh, Finding Legal Reordering Transformations Using Mappings. *Seventh Int'l. Workshop on Languages and Compilers for Parallel Computing Lecture Notes in Computer Science*, Vol. 892 Ithaca, New York Springer-Verlag. (August 1994).
4. V. P. Krothapalli and P. Sadayappan, Removal of Redundant Dependences in DOACROSS Loops with Constant Dependences. *Proc. of the Third ACM SIGPLAN Symp. on PPPP*, pp. 51–60 (July 1991).
5. Ding-Kai Chen, Compiler Optimizations for Parallel Loops With Fine-Grained Synchronization. Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1994. Also available as CSRD Report 1374.
6. Wayne Kelly and William Pugh, A Unifying Framework for Iteration Reordering Transformations. *Proc. of the IEEE First Int'l. Conf. on Algorithms and Architectures for Parallel Processing*, Brisbane, Australia (April 1995).
7. S. P. Midkiff and D. A. Padua, Compiler Algorithms for Synchronization. *IEEE Trans. on Computers*, **C-36**(12):1485–1495 (1987).
8. S. P. Midkiff and D. A. Padua, A Comparison of Four Synchronization Optimization Techniques. *Proc. IEEE Int'l Conf. on Parallel Processing*, pp. II-9–II-16 (August 1991).
9. M. Gupta and E. Schonberg, Static Analysis to Reduce Synchronization Costs in Data-Parallel Programs. *Conference Record of POPL '96: The 23RD ACM SIGPLAN-SIGACT Symp. on PPL* (January 1996).
10. Chau-Wen Tseng. Compiler Optimizations for Eliminating Barrier Synchronization. *Proc. of the Fifth ACM SIGPLAN Symp. on PPPP*, pp. 144–155 (July 1995).