Issues in the compile-time optimization of parallel programs

S.P. Midkiff    D.A. Padua

Abstract – The data-flow and dependence analysis techniques used in today’s optimizing compilers have been designed for sequential source programs. When parallel source programs are to be optimized, the standard techniques are insufficient to ensure the correctness of the transformations. This paper presents eleven examples demonstrating how the standard techniques fail when applied to determinate and non-determinate parallel programs. A more general analysis technique is described that extends the work of Shasha and Snir and guarantees the correctness of optimizing transformations applied to programs with cobegin parallelism. Deadlock avoidance when transforming programs with synchronization is also discussed.

1 Introduction

Parallel language constructs have existed almost as long as sequential programming languages. Such languages often include constructs such as fork and join [4], the cobegin statement [7], and the doall loop [9]. Most manufacturers of parallel machines supply languages with parallel extensions.

When compiling programs written in parallel languages it is desirable to be able to perform the same optimizations as those applied to sequential programs. As the examples in this paper demonstrate, current analysis techniques are insufficient to determine when these optimizations can be safely applied. This problem is overcome today by either not optimizing the program, or by restricting the sharing of data among parallel portions of the program. The first approach diminishes program performance, while the second restricts the class of parallel algorithms that can be implemented.

This paper describes a third approach - a more general analysis technique that allows a compiler to determine when a transformation may safely be applied to a program. This technique works for both determinate and non-determinate parallel programs.

We consider a transformation to be safe if all executions of the transformed program are sequentially consistent [8]. Stated informally, this means that for a given input any result of the transformed program must be possible in the
untransformed program, and if the original program does not deadlock, then neither does the transformed program. However, both programs need not behave identically.

The problem of optimizing parallel programs has been discussed briefly by Shasha and Snir [15], and by Dubois, Scheurich, and Briggs [5]. Shasha and Snir develop an analysis technique for straight-line code that identifies pairs of accesses to shared data whose order cannot be changed by the hardware executing the program or by transformations to the program during compilation. The solution presented here extends their technique to programs with loops.

After dealing with architectural issues, notation, and definitions in Section 2, eleven examples of optimizing transformations that would be incorrectly applied using current analysis techniques are given in Section 3. Section 4 then describes a more general analysis technique.

2 Definitions and notation

A portion of a program that must execute sequentially on a single processor is a thread. In this paper, threads are created using the cobegin statement, and it will be assumed that cobegins are the only parallel construct in source programs. Its syntax is:

\[
\text{cobegin} \\
\text{stmt-list} \\
\ldots \\
\text{stmt-list} \\
\text{coend}
\]

where \text{stmt-list}, a list of statements, is a thread.

Two synchronization statements, set\( (r) \) and wait\( (r) \), are used. The set statement sets bit \( r \). The wait\( (r) \) statement waits until \( r \) is set before completing. If \( r \) is already set, the wait statement completes immediately. Multiple wait statements may wait on a single bit \( r \).

If a statement \( Sp \) is nested in a do loop, an instance of the statement is its execution within an iteration of the loop. An instance of \( Sp \) is denoted \( Sp(i) \), where \( i \) is the value of the index variable for the surrounding loop. If a statement executes exactly once, then the single execution is the sole instance of the statement and is denoted \( Sp \). For example, in the program of Figure 1, \( S1(4) \) is the execution of \( \text{wait}(r(i-1)) \) where \( i = 4 \).

A conflict exists between two statement instances \( Sp(i) \) and \( Sq(j) \) if both access the same memory location and at least one access is a write. This relation is denoted \( Sp(i) \sim Sq(j) \) or \( Sq(j) \sim Sp(i) \).

\footnote{Threads are also called segments in related works, e.g., \cite{15, 14}.}
A dependence \cite{3, 12} exists between two statement instances if they conflict, and one instance will access the memory location before the other. If \( Sp(i) \land Sq(j) \) and \( Sp(i) \) accesses the location first, the \( Sq(j) \) is dependent on \( Sp(i) \). Graphical representations of conflicts and dependences will be discussed in Section 4.

Programs are either determinate or non-determinate. In this paper, a determinate program is one in which all conflicts are ordered, i.e., are dependences. A non-determinate program is one in which one or more conflicts are not ordered. A more complete discussion of determinacy and non-determinacy can be found in \cite{6}.

3 Examples of the problem

This section presents examples showing the problems that arise when attempting to apply compiler optimizations to parallel programs. In each example, the transformation would be valid if the thread were part of a sequential program. Since the program is parallel the same transformation may allow outcomes of the program execution that were not possible with the original program. This demonstrates that violating the assumption that the source program is sequential alters the conditions under which a transformation can be made.

A transformation can violate sequential consistency in two ways. The first way is if two or more variables in the transformed program can assume sets of values that are impossible in the original program, and the values are observable. Examples of this are shown in Sections 3.1 and 3.2. For each example, an assertion is given that is true after the execution of the original program segment, but is no longer true after executing the transformed program segment. Also shown is an execution sequence that is possible in the transformed program and that contradicts the assertion about the sequential program. The solution examples in Section 4 will refer to these sequences. The second way that sequential consistency can be violated is if the program no longer terminates after a transformation is applied. An example of this is shown in Section 3.3.

3.1 Determinate program examples

The examples in this subsection show that compiler optimizations applied to determinate parallel programs, using the criteria for sequential programs, can result in incorrect programs. Since the set and wait instructions can be replaced with busy-wait, a strategy of not optimizing code in the presence of explicit synchronization is problematic since explicit synchronization operations can be replaced by busy-wait synchronization, which is difficult, if not impossible, to detect at compile time.
Before | After
---|---
\(x(1:n) = 0\) | \(x(1:n) = 0\)
cobegin | cobegin
do i = 1, n | do i = 1, n
S1: \(\text{wait}(r(i-1))\) | S1: \(\text{wait}(r(i-1))\)
S2: \(\ldots = x(i-1)\) | S2: \(\ldots = x(i-1)\)
end do | ...
T2: do j = 1, n | if (j .lt. n)
\(\ldots\) | end do
S3: if (j .lt. n) | S3: a(i) = x(i)
\(\text{a}(j) = x(j)\) | \(\text{a}(j) = x(j)\)
end do | do k = 0, n
\(\text{a}(j) = x(j)\) | S4: x(k) = 1
S5: S5: set(r(k))
do k = 0, n | end do
S4: x(k) = 1 | coend
S5: set(r(k)) | coend
coend

assertion: \(a(i) = 1, 1 \leq i \leq n - 1\)
a contradicting sequence: \(S3(2), S4(2), S5(2), S1(3)\)

Figure 1: Loop fusion can introduce non-determinacy

### 3.1.1 Loop fusion

Loop fusion is a transformation that merges two loops into a single loop. It is safe in sequential programs whenever variable accesses involved in dependences spanning the two loops occur in the same order before and after loop fusion. The two loops in the first thread of the determinate program of Figure 1 can be fused using the criteria for sequential programs. If the loops are fused, however, the resulting program is no longer determinate.

In the original program the synchronization performed by \(S5\) and \(S1\), along with the \(j\) loop executing after the \(i\) loop, enforce an order between \(S4(k)\) and \(S3(k)\). After fusing the \(i\) and \(j\) loops, it is no longer true that \(S3(j)\) must execute after \(S1(j - 1)\), and so it is no longer true that the conflict \(S4(k)\) is ordered. Therefore the program is non-determinate.
Before | After
---|---
\texttt{cobegin} | \texttt{cobegin}
\texttt{T1: ...} | \texttt{T1: cobegin...}
\texttt{S1: a = b} | \texttt{S1: a = b}
\texttt{S2: \texttt{wait}(s)} | \texttt{S4: \texttt{set}(s)}\texttt{wait}(s)
\texttt{S3: c = d} | \texttt{S3: c = d} \texttt{set}(s)
\texttt{T2: d = e} | \texttt{T2: d = e}
\texttt{coend} | \texttt{coend} \texttt{coend}

\text{assertion:}\ c = d = e
\text{a contradicting sequence:}\ S3, T2, S4, S2

Figure 2: Concurrentizing a determinate parallel program

### 3.1.2 Concurrentization of straight-line code

In the program of Figure 2, the \texttt{wait}(s) statement forces \texttt{S3} to read a value for \(d\) after \texttt{S4} executes. If analyzed separately, thread \texttt{T1} can be parallelized. If this is done, however, the \texttt{wait} statement no longer controls the execution of \texttt{S3} and the program is no longer determinate. Therefore \texttt{T1} cannot be analyzed in isolation, and the analysis for the program must consider the memory accesses of threads \texttt{T1} and \texttt{T2} together.

### 3.1.3 Concurrentization of a loop

If a source program is sequential, a loop may be made concurrent if all dependences that extend across iterations of the loop are synchronized. In Figure 3 the loop contains no cross iteration dependences, and it appears that concurrentization is correct\(^2\). The assertion and execution sequence of Figure 3 shows the concurrentization to be incorrect.

### 3.1.4 Code hoisting

In a sequential program it is possible to move the computation of an expression to another location in the program if two conditions are met. First, all input

\(^2\text{The semantics of the }\text\texttt{do all}\text{ loop are that every iteration is a thread, and iterations may execute in any order.}\)
variables to the expression must have the same value in the new location as in
the original location. Second, no read of the computed variable may receive a
different value than in the original program. If only the code in thread $T_1$ of
the original program of Figure 4 is considered, the transformation appears to
be correct. The contradiction of the assertion shows that the transformation is
not correct.

It can be argued that the presence of synchronization implies that no at-
ttempts should be made to optimize the code that might be affected by the
synchronization. As the program in Figure 5 shows, doing so is sub-optimal.

3.1.5 Dead code elimination

If a program is sequential, any code that cannot be reached in any execution of
the program or that produces no changes in memory that affect later statements
is considered dead code and can be removed from the program. By this defi-
nition, the if statement in the original program of Figure 6 can be considered
dead code\(^3\) and removed. This happens, for example, when parallel programs
with busy-wait constructs are submitted to the Alliant Fortran compiler\(^2\).

\(^3\)A compiler doing good flow of control analysis might consider this if statement an infinite
loop. Were the source program parallel, it might assume the code was used for busy-waiting
synchronization and treat it as such. The current problem then reduces to the problem of
optimizing in the presence of explicit synchronization.
Figure 4: Code hoisting introducing non-determinacy
Figure 5: Valid code motion and dead code elimination in the presence of synchronization
### Before

c(1:n) = 0

```c
cobegin
do i = 1,n

... S1:
   if (c(i) .eq. 0)
      goto S1
   b(i) = c(i)
   end do

\ S2:
   do j = 1,m
      \ end do

S3:
   c(j) = 1
   end do
coend
```

### After

c(1:n) = 0

```c
cobegin
do i = 1,n

... S2:
   b(i) = c(i)
   end do

\ S3:
   c(j) = 1
   end do
coend
```

assertion: b(i) = c(i) = 1

a contradicting sequence: S2(1), S3(1), S1(1)

---

The program after dead code elimination is clearly incorrect since the synchronization was removed.

### 3.2 Non-determinate program examples

#### 3.2.1 Loop concurrentization

In Figure 7 the loop contains no cross iteration dependences, and it appears that concurrentization is correct. If this is done, however, the semantics of the program are changed, as shown by the execution sequence.

#### 3.2.2 Loop fusion

If the program of Figure 8 were sequential, the loops in thread T1 could be fused since they share no data. The execution sequence of Figure 8 demonstrates that the transformation does not preserve sequential consistency.

#### 3.2.3 Code hoisting

If the original program of Figure 9 were sequential, then the code motion performed on statements S2 and S3 would be incorrect. As can be seen, this transformation does not preserve sequential consistency.
**Before** | **After**
---|---
\begin{align*}
\text{n} &= 0 \\
&\text{cobegin} \\
&\text{do } i = 1, m \\
&\text{end do}
\end{align*} & \begin{align*}
\text{n} &= 0 \\
&\text{cobegin} \\
&\text{do all } i = 1, m \\
&\text{end do}
\end{align*} \\
\begin{align*}
S1: &\quad b(i) = n \\
&\text{end do} \\
&\text{end do} \\
&\text{\textbf{\textbackslash \textbackslash}}} \\
S2: &\quad n = 1 \\
&\text{coend}
\end{align*} & \begin{align*}
S1: &\quad b(i) = n \\
&\text{end do} \\
&\text{end do} \\
&\text{\textbf{\textbackslash \textbackslash}}} \\
S2: &\quad n = 1 \\
&\text{coend}
\end{align*} \\
assertion: &\quad b(i) = 1 \Rightarrow b(i + 1) = 1 \\
a contradicting sequence: &\quad S1(2), S2, S1(1)

Figure 7: Loop concurrentization

**Before** | **After**
---|---
\begin{align*}
&\text{cobegin} \\
T1: &\quad \text{do } i = 1, 100 \\
S1: &\quad a(i) = 0 \\
&\text{end do} \\
&\text{do } i = 1, 100 \\
S2: &\quad b(i) = 1 \\
&\text{end do} \\
&\text{\textbf{\textbackslash \textbackslash}}} \\
T2: &\quad \text{do } j = 1, 100 \\
S3: &\quad b(j) = 0 \\
S4: &\quad a(j + 1), = 1 \\
&\text{end do} \\
&\text{end do} \\
&\text{coend}
\end{align*} & \begin{align*}
&\text{cobegin} \\
T1: &\quad \text{do } i = 1, 100 \\
S1: &\quad a(i) = 0 \\
&\text{end do} \\
S2: &\quad b(i) = 1 \\
&\text{end do} \\
T2: &\quad \text{do } j = 1, 100 \\
S3: &\quad b(j) = 0 \\
S4: &\quad a(j + 1), = 1 \\
&\text{end do} \\
&\text{coend}
\end{align*} \\
assertion: &\quad b(i) = 0 \Rightarrow a(i + 1) = 1 \\
a contradicting sequence: &\quad S2(1), S3(1), S4(1), S1(2)

Figure 8: Loop fusion
Before | After
--- | ---
```
cobegin
T1: do i = 1,100
    S1: c(i) = i
        if (...) then
            S2: t = i
                ... 
                else
                S3: t = i
        end if
    end do
end
\ \ 
T2: t = 0
S4: c(100) = 0
```

cobegin
S2: t = 1
S3: do i = 1, 100
    S1: c(i) = i
        if (...) then
    S2: t = i
        ... 
        else
        S3: t = i
    end if
    end do
end
```
```
assertion: c(100) = 1 \Rightarrow t = 1
a contradicting sequence: S2, T2, S4, S1(100)
```

Figure 9: Code hoisting

### 3.2.4 Common sub-expression elimination

Common sub-expression elimination is an optimization that finds two expressions that always have the same value, and eliminates the computation of one expression. The value of the computed expression is then substituted for the eliminated expression. An example of common sub-expression elimination can be seen in Figure 10. In thread T1 a(i) and d(i) appear to receive the same value: \( \sin(b(i)) \). Since the \( \sin \) function is expensive, assigning \( d(i) \) the value of \( a(i) \) should result in a more efficient program. If the code of thread T2 is ignored, the transformation is correct, since the value of \( b(i) \) is unchanged from statement \( S1 \) to statement \( S3 \). The execution sequence of Figure 10 shows a sequentially inconsistent execution that can result.

### 3.2.5 Loop distribution and vectorization

Loop distribution involves taking the innermost loop(s) of a program and replicating it for each strongly connected component\(^4\) in the original loop. Loop distribution causes all instances of statements in a strongly connected component of one copy of the distributed loop to execute before any instances of a

\(^4\)The strongly connected components of a graph are subgraphs such that any node in the subgraph can reach, and be reached by every other node in the subgraph, and if node \( n \) can reach a node in a subgraph \( G' \), and some node in \( G \) can reach \( n \), then \( n \) is in the subgraph \( G' \).
Before | After
---|---
\[ b(1:100), 3(1:100) = 0 \] | \[ \]
\begin{align*}
\text{cobegin} \\
T1: & \quad \text{do } i = 1,n \\
S1: & \quad a(i) = \sin(b(i)) \\
S2: & \quad e(i) = f(i) \\
S3: & \quad d(i) = \sin(b(i)) \\
\text{end do} \\
\text{\textbackslash \textbackslash} \\
& \quad \text{do } j = 1,n \\
S4: & \quad b(j) = 1 \\
S5: & \quad f(j) = 1 \\
\text{end do} \\
\text{coend}
\end{align*}

\text{assertion: } e(i) = f(i) = 1 \implies d(i) = \sin(1)

a contradicting sequence: \( S1(1), S4(1), S5(1), S2(1), S3(1) \)

Figure 10: Common sub-expression elimination

statement in the strongly connected component of following copies of the distributed loop. If each assignment statement is surrounded by a different copy of the distributed loop, the effect on the execution order of the statement instances is the same as vectorizing the loop. In sequential programs, loop distribution is correct if the dependence graph for the program is acyclic.

An example of loop distribution can be seen in Figure 3.2.5. A loop before distribution makes up the body of thread \( T1 \) in the original program. If the code in thread \( T2 \) were not present, the distribution would be correct, since the data dependence graph for the undistributed loop is acyclic. The execution sequence shows a sequentially inconsistent execution that can result.

### 3.3 A Deadlock example

The example of this section shows that transforming programs that contain synchronization can lead to deadlock. The loops of thread \( T1 \) of the original program of Figure 12 are fused, yielding a deadlocking program. The accompanying graphs give the execution order of some of the statement instances in the program before and after the transformation.

Depending on the code substituted for the ellipses, the program may be determinate or non-determinate. If conflicting accesses are not ordered by the synchronization are substituted for the ellipses, the program will be non-determinate. Otherwise it will be determinate.
\begin{align*}
\text{Before} & \quad \text{After} \\
\begin{array}{ll}
a(1:n), e(1:n) = 0 & a(1:n), e(1:n) = 0 \\
b(1:n), c(1:n) = 0 & b(1:n), c(1:n) = 0 \\
\text{cobegin} & \text{cobegin} \\
T1: \quad \text{do } i = 1, n & T1: \quad \text{do } i = 1, n \\
S1: \quad a(i) = b(i) & S1: \quad a(i) = b(i) \\
S2: \quad e(i) = c(i + 1) & S2: \quad e(i) = c(i + 1) \\
\text{end do} & \text{end do} \\
\text{} & \text{} \\
\text{coend} & \text{coend} \\
\end{array}
\end{align*}

\text{assertion: } e(i) = 1 \Rightarrow a(i + 1) = 1
\text{a contradicting sequence: } S1(2), S4(1), S3(2), S2(1)

Figure 11: Loop distribution
Graphs depict the execution order of the programs

Figure 12: Loop fusion leading to deadlock

Deadlock may result from any transformation that reorders synchronization statements.

4 Analysis for parallel programs

The transformations in Section 3 produce programs that are not sequentially consistent because they alter the execution order of stores and fetches in the original program, and the final value of the variables in the program record this fact. When transforming sequential programs, honoring all data dependences ensures that any alteration in the order of fetches and stores is not detectable when the program terminates. With parallel programs additional constraints may apply, and a more general analysis is needed.

The analysis, to be described below, operates either explicitly or implicitly on a graph that contains every statement instance of the program being compiled. If the portion of the program being analyzed contains no loops, then a graph
containing all statement instances of interest can be built, and the method from [15, 14], described in Section 4.1, should be used. If the program being compiled contains loops with bounds that are unknown at compile time, the number of statement instances will also be unknown and the method of Section 4.2 should be used. This method represents in summary form all statement instances. Finally, if the program being compiled contains loops whose bounds are known at compile time, then either method may be used, although when the number of statement instances is large, the method of Section 4.2 could be preferable.

4.1 Analyzing programs without loops

If the program under consideration has no loops, the results of Shasha and Snir, [15, 14] (briefly described here), can be used directly. These results were primarily developed to determine when multiple memory requests generated by a thread can safely be outstanding in a machine with a non-determinate memory network.

An instance-level conflict graph (or i-level conflict graph) is a graph whose nodes are statement instances connected by directed arcs representing the execution order of statement instances, and undirected edges representing conflicts between statement instances. The directed arcs are program arcs and the undirected edges are conflict edges. All transitive program arcs are included, although they are omitted from the pictorial representations of the graphs in this paper. A mixed cycle is a cycle that contains both conflict edges and program arcs. A minimal mixed cycle is a mixed cycle $C$ such that no other mixed cycle $C'$, whose nodes are a subset of the nodes of $C$, exists, or if such a cycle $C'$ does exist, then all the nodes of $C'$ are contained within the same thread. Figure 13 gives examples of these terms.

Two results of Shasha and Snir summarize this method of analysis.

**Result 1** If the execution order represented by the program arcs contained in all mixed cycles is enforced, then the execution of the program is sequentially consistent,

and

**Result 2** Any set of program arcs whose enforcement ensures sequential consistency is a superset of the program arcs contained in minimal mixed cycles.

Result 1 gives a conservative condition for ensuring sequential consistency, whereas Result 2 gives a minimal condition.

In each example of the previous section, an i-level conflict graph containing the relevant statement instances shows that one or more program arcs involved in a minimal mixed cycle is not being enforced after the transformation is performed. Portions of the i-level graph for each of the examples of Figures 1 through Figure 11 are shown in Figures 14 through 23.
cobegin
S1:    a = b
S2:    c = d
\/
S3:    d = f
S4:    b = e
\/
S5:    f = h
S6:    e = g
coenend

A program

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{collision_graph}
\caption{An i-level collision graph}
\end{figure}

The i-level collision graph

S1, S2, S3, S5, S6, S4, S1

A mixed cycle

S1, S2, S3, S4, S1

A minimal mixed cycle

Figure 13: An i-level collision graph
Figure 14: I-level graph for the program of Figure 1

Figure 15: I-level graph for the program of Figure 2

Figure 16: I-level graph for the program of Figure 3
Figure 17: I-level graph for the program of Figure 4

Figure 18: I-level graph for the program of Figure 6

Figure 19: I-level graph for the program of Figure 7

Figure 20: I-level graph for the program of Figure 8
Figure 21: I-level graph for the program of Figure 9

Figure 22: I-level graph for the program of Figure 10

Figure 23: I-level graph for the example of Figure 11
Figure 24: The $s$-level collision graph and maps for the program of Figure 11

4.2 Analyzing programs with loops

The method of this section uses a statement level conflict graph ($s$-level graph) to compactly represent an unbounded number of statement instances. An $s$-level graph has statements as nodes, joined by directed arcs representing execution order and pairs of directed arcs representing conflicts. A map is associated with each arc and describes the particular statement instances that are members of the relation represented by the arc. An instance of an arc is a copy of the arc for two particular statement instances on the associated $i$-level graph. $S$-level graphs, maps, and their uses are discussed further in [10]. Figure 24 gives the $s$-level conflict graph for the loop distribution example of Section 3.2.5.

Since the map attached to each arc describes all pairs of statement instances associated with the arc, the collection of maps attached to the arcs in a path through the graph describes the statement instances visited on that path. Thus the $s$-level graph gives a compact representation of a (possibly unbounded) $i$-level graph and the paths through that graph. A mixed path in the $s$-level graph corresponds to a mixed cycle in the associated $i$-level graph only if a solution exists to the system of equations and inequalities formed by the maps associated with the arcs in the cycle.

For example, consider the cycle implied by the execution sequence of Figure
which includes the cycle shown in Figure 23. The system generated by this cycle is:

\[
1 \leq i, i', i'', j, j' \leq n \\
j = i - 1 \\
j' = j + k_3, k_3 \geq 1 \\
i' = j' - 1 \\
i'' = i' + k_1, k_1 \geq 1 \\
i = i''
\]

Using integer programming techniques [13], it can be seen that this system has solutions, and therefore that a cycle does exist in the implicitly defined i-level graph. If dependence analysis techniques are used, the maps attached to program arcs define the direction vector for that program arc.

One solution is that shown by the execution sequence of Figure 11, and the cycle in Figure 23:

\[
i = i'' = 2, i' = 1, k_1 = 1 \\
j = 1, j' = 2, k_3 = 1
\]

Thus, there exists at least one cycle in the i-level graph that contains an instance of the \(\rho_3\) arc. The incorrect execution order discussed in Section 3.2.5 violates the order implied by one instance: \(S2(1) \rightarrow S1(2)\). Thus the conservative solution, implied by Result 1, requires that this arc be honored. In this example the cycle is minimal, and by Result 2 each violation of the order implied by an instance of the \(\rho_3\) arc leads to an incorrect execution.

### 4.3 Applying the analysis

When determining the correctness of applying a transformation to a thread of a non-determinate program, analyzing only the thread to be transformed is insufficient. Instead, the intra-thread data access information needs to be merged with sequencing information derived from the more general analysis. Program arcs that must be honored can be represented in several ways, such as arcs on a dependence graph for the thread, or by modifying the UD-chains and IN-OUT sets[1] for the thread.

In Figures 14 through 23, portions of the i-level graphs for each of the examples in Section 3 are given. Were s-level graphs constructed for these programs, each cycle in the i-level graph would correspond to a solution to the system of equations generated by some mixed cycle in the s-level graph.
4.4 Some approximate solutions

Given that the analysis techniques discussed can, in the worst case, be very expensive, some guidelines that allow limited optimization without performing the analysis are useful.

If the access order of variables is not changed by a transformation, then it is not possible for the transformation to violate a program arc involved in a minimal mixed cycle, and the analysis of Section 4 are unnecessary. One such transformation is simple-minded register allocation where values are fetched into a register immediately before the execution of the operation where they are used and spilled immediately afterwards. Another transformation of this type is loop blocking [12].

Even if the access order of variables within a thread is changed, a problem can only arise when a program arc involved in a minimal mixed cycle is violated. If a memory access does not conflict with an access in another thread, then the node in the i-level graph representing that access cannot be part of a minimal mixed cycle, and therefore no program arc involved in a minimal mixed cycle can be violated. Thus, the transformation is valid if it honors intra-thread data-flow and dependence constraints.

Finally, the access order specified by a program arc in a minimal, mixed cycle can only be violated if a memory access to a shared variable is moved before or after a memory access to another shared variable. A transformation that alters the memory access order of a shared variable relative only to non-shared variables is correct, assuming that intra-thread data-flow and dependence constraints are observed. Thus, for example, placing a shared variable in a register is always valid so long as the lifetime of the value does not extend beyond any stores and fetches to other shared variables.

4.5 Deadlock avoidance

In Section 3.3 it was shown that deadlock can result from transforming programs containing synchronization. In this section, a method to determine if a given statement ordering leads to deadlock is outlined. A modification of this method can be used to determine all statement orderings that might lead to deadlock.

To detect potential deadlock a deadlock graph, similar to a conflict graph, is used. The nodes are statements or statement instances, depending on the level of the graph. Two types of arcs can be found on the graph. The first, synchronization arcs, represent execution orders imposed by synchronization. If the order of execution is known at compile time, a directed arc reflecting this order is added to the graph. If the order of execution enforced by the synchronization is not known, a pair of directed arcs is added to the graph.

If the synchronization is provided by predefined synchronization instructions, determining the order imposed should be straightforward. If busy-wait synchronization is used, then the busy-wait in one thread waits for certain values to
be assigned to variables in other threads. Therefore, the synchronization arcs will be from assignments to those variables to the boolean expression of the if statement that controls the busy-wait. For example, the deadlock graph for the program of Figure 3 would have a synchronization arc from the $S4$ assignment to the if statement, $S1$. Finally, shared variables may be accessed within a subroutine or function call. In the absence of inter-procedural analysis (or user assertions), it is necessary to assume that synchronization is occurring and that the order imposed by the synchronization is unknown.

The second type of arc is the program arc, which is like the program arc on a conflict graph. The graphs of Figure 12 are examples of deadlock graphs.

For deadlock to occur, it is necessary that a cycle exists in the deadlock graph. Since the goal is to determine if a statement order can lead to deadlock, only cycles that include both program arcs and synchronization arcs are of interest. The graph for the original program of Figure 12 contains no cycles, thus the original program does not deadlock. In the graph for the program after the loops have been fused, a cycle exists and therefore deadlock will occur.

It can happen that the graph for the original program is cyclic. This results from any of three causes. The first is that the original program deadlocks. The second is that inaccuracies in determining the order imposed by synchronization has included extraneous synchronization arcs in the graph. The third is possible only if a statement level deadlock graph is used. Cycles may appear to exist because the complexity of a system of equations may require that a solution be assumed to exist even when none does. In any case, if the statements involved in such cycles execute in the same order relative to one another, the deadlock characteristics of the program do not change.

5 Conclusions and future work

Designers of parallel languages have either ignored the problem of optimizing programs written in those languages or have overcome the problem by prohibiting the sharing of data among threads of the program executing in parallel. As the examples of this paper demonstrate, the problem can only be ignored if parallel programs are not optimized - a course of action that may incur a large performance penalty. Attempts to legislate the problem away by placing restrictions on data sharing lead to what we feel is an excessively restrictive programming model.

Using the methods of [14, 15] and the extensions described in this paper, it is possible to perform an analysis on parallel programs that describes allowable reorderings of memory accesses. If deadlock is not a concern, the constraints uncovered by this analysis can be described in traditional dependence or data-flow form and therefore be cleanly integrated into an optimizing compiler.

Much work remains to be done. The techniques described in this paper are being implemented in the Delta program transformation system [11].
techniques presented assume that all threads in a program can be represented on the conflict graph. If parallelism is expressed as a parallel loop with unknown loop bounds instead of as a cobegin, this is not possible. Currently, solutions only exist for restricted classes of conflict maps when loop parallelism is present. Finally, it is possible to ensure sequentially consistent executions by enforcing an order on conflict arcs instead of program arcs. This is discussed briefly in [10, 15].

6 Acknowledgements

The authors would like to thank Ron Cytron and the referees for their helpful comments. This work was supported by NSF Grant MIP-8410110, DOE Grant DE-FG02-85ER25001, and a grant from the Control Data Corporation.

References


