Concurrency Analysis in the Presence of Procedures Using a Data-Flow Framework*

Evelyn Duesterwald Mary Lou Soffa

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract - Although the data-flow framework is a powerful tool to statically analyze a program, current data-flow analysis techniques have not addressed the effect of procedures on concurrency analysis. This work develops a data race detection technique using a data-flow framework that analyzes concurrent events in a program in which tasks and procedures interact. There are no restrictions placed on the interactions between procedures and tasks, and thus recursion is permitted. Solving a system of data-flow equations, the technique computes a partial execution order for regions in the program by considering the control flow within a program unit, communication between tasks, and the calling/creation context of procedures and tasks. From the computed execution order, concurrent events are determined as unordered events. We show how the information about concurrent events can be used in debugging to automatically detect data races.

1. Introduction

Concurrent constructs, such as tasks and parallel loops, are introduced into programming languages to enable the explicit expression of parallelism. However, the improper use of such constructs may create program access anomalies, also termed *data races*.

* This work was partially supported by the National Science Foundation under Grant CCR-8801104 to the University of Pittsburgh.

Data races introduce the problem of nondeterminism, and the existence of nondeterminism in a program is difficult and expensive to detect. Static concurrency analysis addresses the problem of automatically detecting access anomalies that lead to nondeterminism by identifying possible concurrent events in a program prior to execution. Although data-flow analysis techniques have been developed to statically determine the potential concurrency among statements, current data-flow techniques have not incorporated procedures into the analysis. [3-5, 14] However, today's emphasis on top down design mandates the inclusion of procedures and their interactions with concurrent constructs into concurrency analysis.

We present a data-flow based analysis technique for determining concurrent statements in programs that use explicit concurrent events in the form of tasks, with synchronous intertask communication through *entry call* and *accept* commands as defined by the rendezvous concept of Ada [7]. Our analysis technique supports the interaction of procedures, tasks, sequential loops and conditionals. Procedures can be called recursively, and tasks can be created dynamically as the program executes. We have developed a system of data-flow equations to express a partial execution ordering for regions in the program in a *happened-before* and *happened-after* relation. The solution of the data-flow equations determines statements that execute **before** and **after** a statement. Statement ordering is enforced by various mechanisms in a concurrent program. On a local level, the *control flow* among statements prescribes an ordering of statements within a task or procedure. *Task synchronization* commands enforce statement ordering

among statements across task boundaries at synchronization points. The ordering of statements in procedures and tasks is also influenced by the *calling context* of a procedure and the *creation context* of a task. Finally, the potential occurrence of *parallel instances* of the same procedure or task can also affect the statement ordering information in that procedure or task.

To determine the partial execution order of statements, sets of data-flow equations are defined that express the above ordering restrictions in a concurrent program. The equations are solved iteratively on a graphical representation of the program, the *Module Interaction Graph* (MIG), which was developed as part of this work. Using the partial execution order, statements that can execute in parallel are determined. We show how the information about parallel statements, combined with alias information, is finally used to detect potential data races. We have adapted the work by Callahan, Kennedy and Subhlok developed for low level synchronization in parallel loops and parallel case statements [4,5] to higher forms of concurrent constructs in order to be used as a basis for our efficient incorporation of procedures.

The problem of the exact static determination of concurrent events has been shown to be NP-complete [16]. Static data-flow analysis approximates the computation of concurrent events by detecting all potentially concurrent events in polynomial time but also reports spurious concurrency. We address this imprecision in our technique by careful analysis. In the remainder of this paper, we first discuss the effect of procedures on the concurrency in a program and then give an overview of our concurrency analysis algorithm. Details of the analysis are given in Section 4. Section 5 discusses how we maintain precision throughout the analysis. Data race detection and other applications of the computed execution order are described in Section 6. Section 7 gives a conclusion and discusses related work.

2. The Effects of Procedures on Concurrency

The problems that arise in detecting concurrent events when procedures are introduced into a concurrent program are mainly due to the dynamic activation of such units. With procedures, task activation does not solely follow a static fork-join mechanism. Instead task activation may occur dynamically as a side effect of procedure execution, resulting in a potentially unlimited nesting of parallelism. Also, a procedure may be called from various sites, resulting in different potential concurrent events with each site. Thus besides being able to determine the parallelism within each procedure, the major challenge lies in analyzing the parallelism that results from the interaction of procedures and tasks. Lastly, considerable care must be taken to provide information that is precise enough to be of practical use; that is, the information cannot be overly conservative. Inherent to static data-flow analysis, the precision of the data-flow information is limited by the assumption that all execution and synchronization paths in the program are feasible.

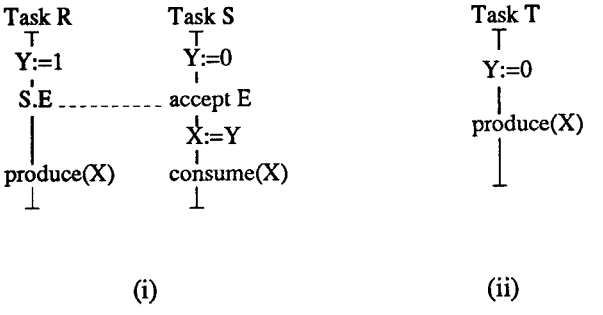


Fig. 1: Interacting tasks

To illustrate the problems, consider the program fragment in Fig. 1 (i), which shows two tasks, R and S, that communicate through the entry E. The communication point reveals that the definitions of Y in tasks R and S may be concurrent, but the use of Y in task S is ordered with either definition. The example also shows that, considering the synchronization between tasks R and S, the calls to procedures *produce* and *consume* may execute concurrently. Without analyzing possible interactions of the procedures it must be assumed that every statement in *produce* may execute concurrently with every statement in *consume*. Our technique incorporates an analysis of procedure and task interactions that may reveal that the parallelism at the call sites is actually far more limited than a conservative approach would assume. Thus, our analysis may reveal that statements in *produce* and *consume* cannot execute concurrently.

The situation becomes more complicated when a procedure or a task is activated concurrently with itself as depicted in Fig. 1, where task T shown in Fig. 1 (ii) executes concurrently with the two tasks in Fig.

1 (i). Procedure *produce* can be called from tasks R and T simultaneously, resulting in portions of *produce* executing in parallel with themselves. In our technique we analyze the activation context of a procedure or a task in order to detect parallel instances of the same statement.

3. Overview

The core of the concurrency analysis technique is the computation of a partial execution order of statements, which is determined using a data-flow framework. The technique first constructs the *Module Interaction Graph* (MIG) for a program by grouping statements into regions such that if any one statement in a region r can execute in parallel with another statement s, all statements in r can execute in parallel with s. A partial execution order among regions is a relation defined as follows:

Definition: Regions r and s are *ordered* if for every instance of r and s, either r completes execution *before* s starts execution or r starts execution *after* s completes.

The analysis to determine the partial execution order is driven by the ordering among regions enforced by control flow, synchronization and procedure and task activation. The ordering by control flow is analyzed locally for each program unit, i.e., each procedure or task. The gathered information is then propagated across unit boundaries by examining the inter-unit ordering mechanisms, i.e., synchronization and procedure and task activation. Determining the execution order is more complicated in the presence of program units that may execute in parallel with themselves. A unit that may execute concurrently with itself is termed a *parallel unit*. For a parallel unit U, it is possible that each region inside U executes in parallel with any other region in U. Thus, the ordering determined locally for the regions inside U may be invalid. However, a parallel unit cannot be detected unless ordering information is available, as a parallel unit results from unordered activations. We accommodate this difficulty in a two phase algorithm by first assuming that the textual code models the behavior of the program in that only one instance of each procedure and task is ever executing at any one time, i.e., there are no parallel units. The resulting order determined using this assumption is then restricted in a post mortem phase by analyzing the activation sites in the program for parallel units.

The concurrency analysis technique consists of the following phases:

Phase 1: Ordering Assuming no Parallel Units
During the first phase of the analysis we determine for every region r the set *before* (r) and *after* (r) containing regions that are ordered to execute **before** region r and the region that are ordered to execute **after** r, respectively. The sets are computed in three steps based on control flow, synchronization and program unit activation. Data-flow equations solved in each step describe the particular ordering mechanism under investigation.

Step 1: Local Control Flow Analysis
A local order within each unit is determined from the control flow. As control predecessors of a region r execute before r and control successors execute after r, a region is ordered by considering all its control predecessors and successors.

Step 2: Synchronization Analysis
Ordering across unit boundaries is enforced by task synchronization. By examining the possible entry calls and accepts among tasks, the locally computed order is propagated across different tasks in order to achieve a global execution order.

Step 3: Activation Context Analysis
Finally, the activation context of a unit also forms a way of enforcing order among regions, in that regions that occur in *before* sets of all activation sites to a unit are placed in the *before* sets of the unit. Similarly, termination points affect the *after* sets. Thus region ordering can be determined by propagating the activation sequence into the activated unit.

Phase 2: Analysis of Parallel Units
The presence of parallel units requires an update of the previously determined execution order, since the ordering sets determined for each region may not be valid in the case that regions execute in parallel with themselves. Thus the second phase consists of detecting parallel units by analyzing the previously determined order at activation sites and appropriately updating the ordering sets for parallel units.

After the analysis, we have an execution order for all regions. The information is then complete for use in determining concurrent events. For each region r, we detect as concurrent all of those regions

that are not ordered with respect to region r by examining the computed ordering sets for the region r.

4. The Concurrency Analysis

We discuss each of the analysis steps in more detail in this section.

4.1. The Module Interaction Graph (MIG)

The MIG is based on the decomposition of a program unit into a collection of regions. A region forms a maximal single-entry, single-exit group of statements that is uninterrupted by unit (i.e., procedure or task) interactions. In the chosen language model, unit interactions are established by procedure invocation and task creation, and by task communication through entry call and accept commands.

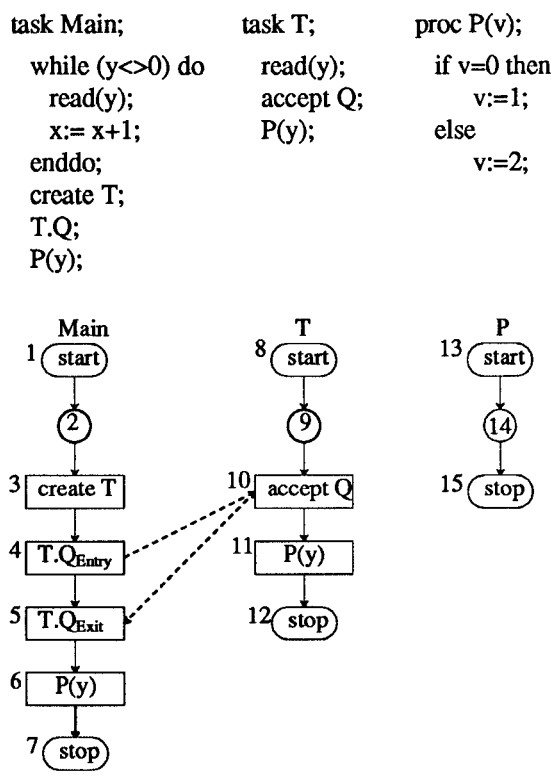


Fig. 2: The MIG for a program example

The construction of the MIG is performed in two steps. The MIG subgraph $G = (R, E_C)$ for each program unit is first constructed and consists of a set of regions R representing sets of textually consecutive statements and a set of control edges E_C. Consecutive statements that do not contain any unit interactions are summarized in a single region. For example, a loop that does not contain any unit interaction commands, such as the loop in task Main in Fig. 2, is represented in a single region (e.g., region 2 in Fig. 2). The subgraphs are then connected by synchronization edges that form the set of edges E_S. The MIG for the entire program $G = (R, E_C \cup E_S)$ is obtained as the union over the subgraphs including the synchronization edges E_S.

Fig. 2 illustrates the MIG for a program example. Each program unit U is enclosed with a special *start* region denoted *U.start* and a *stop* region denoted *U.stop*. Regions in the MIG that are displayed in circles contain sequences of statements that are uninterrupted by unit interactions. Regions containing a unit interaction command are displayed in rectangles.

4.2. Phase 1: Ordering Assuming no Parallel Units

After the MIG is constructed, the partial execution order is computed by iteratively solving a set of data-flow equations over the MIG. Inherent to static analysis techniques, all execution paths in the program are assumed to be feasible, including control as well as synchronization paths. The partial execution order is formulated by two relations *before* and *after* and computed at every region r in the sets *before* and *after*.

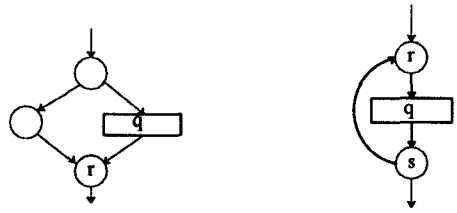
Definition: Given a region r, the set *before*(r) contains regions that, if they execute, complete execution before r starts execution. The set *after*(r) contains regions that, if they execute, start execution after r completes.

We first assume that the textual code models the behavior of the program in that only one instance of each procedure and task is ever actively executing at any one time. Note that multiple instances of a procedure may exist as in recursion but only one is actively executing at any one time. The computation with this assumption is performed in three steps, each step providing an input to the final *before* and *after* sets.

4.2.1. Step 1: Local Control Flow Analysis

In the first step an ordering of the local regions for every unit is computed. The computation is performed for each unit separately with the objective of

determining the local control restrictions of region execution and summarizing them in sets termed $before_{loc}$ and $after_{loc}$.



(i) (ii)

Fig. 3: Regions in a conditional statement and inside a loop

The MIG fragment in Fig. 3 (i) suggests that the local regions executing *before* a region r are the union over r's control predecessors. However, Fig. 3 (ii) that depicts regions inside a loop, shows that control predecessor s of region r can also be a control successor and may execute both before and after region r. It follows that such a region can neither be placed in $before_{loc}(r)$ nor in $after_{loc}(r)$. We accommodate this situation by first computing sets termed *may_before* and *may_after* containing all control predecessors and control successors, respectively. The following data-flow equations illustrate the computation.

$$may\_before(r) = \bigcup_{\substack{s \text{ a control} \\ pred}} (may\_before(s) \cup \{s\}) \qquad (1)$$

$$may\_after(r) = \bigcup_{\substack{s \text{ a control} \\ succ}} (may\_after(s) \cup \{s\})$$

The sets $before_{loc}$ are then obtained by eliminating those control predecessors from *may_before* that are also control successors. The $after_{loc}$ sets are obtained similarly.

$$before_{loc}(r) = may\_before(r) - may\_after(r) \qquad (2)$$

$$after_{loc}(r) = may\_after(r) - may\_before(r)$$

Table 1 shows the $before_{loc}$ and $after_{loc}$ sets for the example of Fig. 2.

Note that in the above treatment of loops a region r inside a loop describes the entire iteration space of the code represented by r. Two regions r and s are considered ordered only if the iteration space of r executes either entirely before or after s. However, a region r inside a loop may be ordered with another region s, such that some iteration instances of r execute before and some execute after s. The analysis can be refined by distinguishing among different iteration instances of a region inside a loop. By distinguishing earlier, the current, and later iterations of a loop, we can compute an execution ordering for groups of region instances. A refined loop analysis that computes an execution order for instances of statements appears in [8].

region	$before_{loc}$	$after_{loc}$	$before_{sync}$	$after_{sync}$
Main				
1	-	2-7	-	10-12
2	1	3-7	-	10-12
3	1-2	4-7	-	10-12
4	1-3	5-7	-	10-12
5	1-4	6-7	8-10	-
6	1-5	7	8-10	-
7	1-6	-	8-10	-
task T				
8	-	9-12	-	5-7
9	8	10-12	-	5-7
10	8-9	11-12	1-4	5-7
11	8-10	12	1-4	-
12	8-11	-	1-4	-
proc P				
13	-	14-15	-	-
14	13	15	-	-
15	13-14	-	-	-

Table 1: Results from step 1 and step 2

4.2.2. Step 2: Synchronization Analysis

After the local region order has been computed for every program unit, the statement ordering imposed by task synchronization is computed by propagating the local $before_{loc}$ and $after_{loc}$ sets across synchronization edges. Thus a further contribution to the sets *before* and *after* is computed, which is expressed by the sets $before_{sync}(r)$ and $after_{sync}(r)$. $Before_{sync}(r)$ contains the regions that, if they execute, are forced by task synchronization to complete before r starts execution. The set $after_{sync}(r)$ is defined similarly.

Consider Fig. 4 and the accept statements in regions 2 and 3. The regions that enter the $before_{sync}$ sets of a unit at synchronization points can be determined as the intersection over synchronization prede-

cessors and their *before* sets. For example, the regions entering $before_{sync}(2)$ are region $r1$ and its *before* set, and the regions entering $before_{sync}(3)$ are the intersection over $r2$, $r3$ and their *before* sets. However, if a synchronization predecessor r is contained in a loop (e.g., $r \in may\_after(r)$) it cannot be assumed that all instances of r complete before the rendezvous takes place. Thus loop regions are prevented from propagation along synchronization edges. The above observations are expressed in the equation for the set *S_before* in equations (3). The set $gen(s)$ in equation (3) refers to the region generated at s. $Gen(s)$ simply contains s unless s is a multiple instance region inside a loop, hence $gen(s) = s - may\_after(s)$.

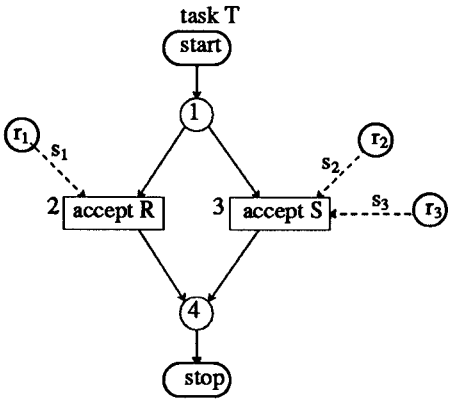


Fig. 4: Task communication

$$before_{sync}(r) = C\_before_{sync}(r) \cup S\_before_{sync}(r)$$

$$C\_before_{sync}(r) = \bigcap_{s \text{ a control pred}} before_{sync}(s)$$

$$S\_before_{sync}(r) = \bigcap_{s \text{ a sync pred}} [before_{loc}(s) \cup before_{sync}(s) \cup gen(s)] \quad (3)$$

$$after_{sync}(r) = C\_after_{sync}(r) \cup S\_after_{sync}(r)$$

$$C\_after_{sync}(r) = \bigcap_{s \text{ a control succ}} after_{sync}(s)$$

$$S\_after_{sync}(r) = \bigcap_{s \text{ a sync succ}} [after_{loc}(s) \cup after_{sync}(s) \cup gen(s)]$$

Once regions have entered the $before_{sync}$ sets of a unit at a synchronization point, they are further propagated throughout the unit along control edges. Consider region 4 in Fig. 4. Only the intersection of $before_{sync}(2)$ and $before_{sync}(3)$ can safely be assumed to execute before region 4. The resulting equation is given for the set *C_before*. Equations (3) show that we obtain the set $before_{sync}$ as the union over *C_before* and *S_before*.

From the first two steps we obtain the first global estimate of the *before* and *after* sets:

$$before(r) = before_{loc}(r) \cup before_{sync}(r)$$

$$after(r) = after_{loc}(r) \cup after_{sync}(r)$$

Table 1 shows the $before_{sync}$ and $after_{sync}$ sets for the example of Fig. 2. Note that the sets for procedure P are empty, since P does not communicate with tasks. The initial estimate of *before* and *after*, as computed by the above equations, is shown in Table 2.

4.2.3. Step 3: Activation Context Analysis

During this next step the ordering imposed by the activation context of a unit U is taken into account by propagating the regions that execute before every activation and the regions that execute after every activation of U into the unit U. The regions that execute before every activation of a unit U are determined as the intersection over all activation sites and their *before* sets. Thus we can determine the set $before(U.start)$ as depicted in equation (4). Similarly, we define the set $after(U.stop)$ based on the termination sites of unit, but we do not include the termination sites themselves in the computation. The termination sites depend on the semantics of procedure and task termination. The termination sites of a procedure are identical to the activation sites and we define the termination sites of a task as the *stop* regions of units that contain a create statement for the task.

$$before(U.start) = \bigcap_{s \text{ an activation site}} [before(s) \cup gen(s)] \quad (4)$$

$$after(U.stop) = \bigcap_{s \text{ a termination site}} after(s)$$

For example, consider Table 2 with the initial estimate of *before* and *after* sets for the program in Fig. 2. Regions that denote an activation site are marked with *. From Table 2 we determine that $before(T.start) = \{1,2,3\}$, since there is only a single activation site to T in region 3. For procedure P, we determine from the two activation sites at regions 6

and 11 that *before* (*P.start*) = {1-6,8-10} $\cap$ {1-4,8-11} = {1-4,8-10}.

region	initial estimate *before*	*after*
task Main		
1	-	2-7,10-12
2	1	3-7,10-12
***3**	1-2	4-7,10-12
4	1-3	5-7,10-12
5	1-4,8-10	6-7
***6**	1-5,8-10	7
7	1-6,8-10	-
task T		
8	-	5-7,9-12
9	8	5-7,10-12
10	1-4,8-9	5-7,11-12
***11**	1-4,8-10	12
12	1-4,8-11	-
proc P		
13	-	14-15
14	13	15
15	13-14	-

Table 2: Initialization for step 3

region	final *before*	*after*
task Main		
1	-	2-15
2	1	3-15
***3**	1-2	4-15
4	1-3	5-7,10-15
5	1-4,8-10	6-7
***6**	1-5,8-10	7
7	1-6,8-10	-
task T		
8	1-3	5-7,9-15
9	1-3,8	5-7,10-15
10	1-4,8-9	5-7,11-15
***11**	1-4,8-10	12
12	1-4,8-11	-
proc P		
13	1-4,8-10	14-15
14	1-4,8-10,13	15
15	1-4,8-10,13-14	-

Table 3: Results from step 3

Once we have determined the activation context for a unit U in the sets *before* (*U.start*) and *after* (*U.stop*) we can further propagate these sets to *before* and *after* sets throughout unit U and other units that interact directly or indirectly with U through procedure calls and task interaction. Here, we can take advantage of the information already computed. The regions that execute before any activation of a unit U clearly also execute before any region that starts execution after any activation of U. The regions that start execution after any activation of U are the regions in *after* (*U.start*). Thus, we can place every region computed in *before* (*U.start*) into *before* sets of regions in *after* (*U.start*). Similarly, we can further propagate regions in *after* (*U.stop*).

For example, when task T from Fig. 2 is processed we place the regions in *before* (*T.start*) = {1,2,3} into the *before* sets of regions in {5-7,9-12} (i.e., regions in *after* (*T.start*)) as shown in Table 3. Similarly, when processing procedure P, we place the regions determined in *before* (*P.start*) = { 1-4,8-10} in the *before* sets throughout P (i.e., regions in *after* (*P.start*)). The propagation of regions in *after* (*U.stop*) of a program unit U is performed analogously.

Note that information is only propagated in one direction - from activation sites to activated units. Here, we take advantage of the redundancy that is present in the *before* and *after* sets of the entire graph. For a pair of regions r and s such that $r \in$ *before* (s), it must be that $s \in$ *after* (r). Thus whenever a region r is added to *before* (s) a *mutual update* is performed by also adding s to *after* (r).

/* determine the activation context */

$$before(U.start) = \bigcap_{s \text{ an activation site}} [\, before(s) \cup gen(s)]$$

$$after(U.stop) = \bigcap_{s \text{ a termination site}} after(s)$$

/* propagation */

for every region $r \in$ *before* (*U.start*) do
  for every region $s \in$ *after* (*U.start*) do
    place r in *before* (s)
    place s in *after* (r) /* mutual update */

for every region $r \in$ *after* (*U.stop*) do
  for every region $s \in$ *before* (*U.stop*) do
    place r in *after* (s)
    place s in *before* (r) /* mutual update */

Fig. 5: Algorithm for the Activation Context Analysis

The activation context analysis iterates over the program units until the information stabilizes. A summary of the individual steps is shown in Fig. 5. Table 3 shows the final *before* and *after* sets after the activation context analysis. If the units are analyzed according to invocation order, i.e., in the order Main, T and P, the activation context analysis proceeds in a single pass. However, in general, recursion may cause a worst case of $O(P^2)$ unit visits, where P is the number of program units.

4.3. Phase 2: Analysis of Parallel Units

During this second phase, we determine the final ordering set $Ord(r)$, the set of regions that, for every execution of region r, are ordered with r. An initial estimate is obtained as the union of the *before* and *after* sets from the previous phase. A special case arises if a region r lies on a branch of a conditional statement. Regions that lie on other the branch will neither occur in $before(r)$ nor in $after(r)$. However, since actually only one branch is executed we can consider regions on both branches as ordered with r and increase the precision by adding all local regions of a unit U to the initial estimate. Thus we obtain $Ord_{init}(r) = before(r) \cup after(r) \cup \{s \mid s \in U\}$. If the program contains no parallel units, the initial estimate is the final value, i.e., $Ord(r) = Ord_{init}(r)$. The Ord_{init} sets for the example in Fig. 2 are shown in the first column in Table 4. However, the existence of parallel units has not yet been taken into account. If a unit U may execute in parallel with itself, it is possible that every region inside U executes in parallel with any other region in U. Thus the order determined in the previous phase, and in particular the ordering based on local control flow, may be invalid and must be updated.

Consider again the example in Fig. 2. Procedure P is called by tasks Main and T simultaneously, resulting in potentially parallel activations of procedure P. The ordering information computed in the previous phase enables us to detect that P is a parallel unit, as the initial *Ord* sets correctly indicate that the two call sites at regions 6 and 11 can execute concurrently (i.e., $6 \notin Ord_{init}(11)$ and $11 \notin Ord_{init}(6)$).

Moreover, if a unit is a parallel unit, the computed ordering sets at the activation sites express the additional parallelism resulting from the parallel execution of the unit; each region that may execute in parallel with an activation site to a unit may also execute in parallel with the unit itself. For example, in Table 4, the Ord_{init} sets at the activation sites to P in regions 6 and 11 both indicate that the sites may execute concurrently with regions in P (i.e., regions 13 through 15 are neither contained in $Ord(6)$ nor in $Ord(11)$). Conversely, if a unit is not a parallel unit, the activation sites contain all unit regions in their ordering sets by the means of mutual updates during the activation context analysis. For example, Table 4 shows that the activation site for task T in region 3 is ordered with all regions of task T.

rg.	initial Ord_{init}	final *Ord*	concurrent
Main			
1	1-15	1-15	-
2	1-15	1-15	-
***3**	1-15	1-15	-
4	1-7,10-15	1-7,10-15	8-9
5	1-10	1-10	11-15
***6**	1-10	1-10	11-15
7	1-10	1-10	11-15
T			
8	1-3,5-15	1-3,5-15	4
9	1-3,5-15	1-3,5-15	4
10	1-15	1-15	-
***11**	1-4,8-12	1-4,8-12	5-7,13-15
12	1-4,8-12	1-4,8-12	5-7,13-15
P			
13	1-4,8-10,13-15	1-4,8-10	5-7,11-15
14	1-4,8-10,13-15	1-4,8-10	5-7,11-15
15	1-4,8-10,13-15	1-4,8-10	5-7,11-15

Table 4: Results from phase 2

In terms of data-flow equations, we express the above observation for updating a parallel unit by viewing the ordering sets at the activation sites as a filter for the orderings of the activated unit. The regions that are ordered with all activation sites of a unit U are expressed in the set $Ord(U)$ and determined as shown in equations (5). $Ord(U)$ is used as a filter for the orderings of regions inside U by eliminating all regions that do not occur in $Ord(U)$, i.e., regions that may execute when U is already activated. The filtering of the initial estimate of *Ord* sets is expressed in the second equation in (5).

$$Ord(U) = \bigcap_{s \text{ an activation site}} Ord(s) \qquad (5)$$

$$Ord(r) = Ord(r) \cap Ord(U) \quad \text{for } r \in U$$

In the example of Fig. 2, we obtain from the two call sites to P in regions 6 and 11 that $Ord(P)$ = {1-4, 8-10}. Filtering the *Ord* sets of regions inside P will eliminate all regions not in $Ord(P)$ and in particular regions 13 through 15. Thus the final *Ord* sets correctly indicate that P may execute concurrently with itself. Note that applying equations (5) to a unit that is not a parallel unit will have no effect. For example for task T, we obtain $Ord(T)$ = {1-15} and filtering the orderings in T with the filter $Ord(T)$ will have no effect. Thus rather than explicitly detecting parallel units, equations (5) are applied iteratively to all units until the information stabilizes. Table 4 shows the final *Ord* sets for the example in Fig. 2.

A special case of a parallel unit arises if a task recursively creates itself. Unlike the case of procedure recursion, the recursive instances of a task all execute concurrently. Thus a recursive task forms a parallel unit with, however, the distinction of a cyclic activation chain. Although we do not discuss task recursion here and restrict recursion to procedures, only minor extensions to our analysis are necessary to accommodate the cyclic activation of a parallel unit formed by a recursive task (see [8] for more detail).

We now conclude the analysis with the following claim.

Claim: (Correctness) Given two regions r and s. Region r may execute concurrently with s, if and only if $r \notin Ord(s)$ or $s \notin Ord(r)$.

Proof: Given the correctness of the system of data-flow equations to compute the *Ord* sets, the claim follows immediately. The correctness of the data-flow equations follows from the discussion given with each set of equations.

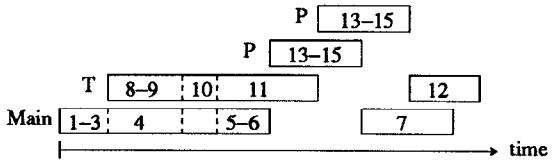


Fig. 6: A possible execution sequence for Fig. 2

Fig. 6 illustrates one possible execution sequence for the program from Fig. 2. The execution sequence shown demonstrates some of the potential parallelism in the example. The complete potential parallelism that was computed in our concurrency analysis is shown in the last column of Table 4.

Using standard iterative data-flow analysis algorithms the concurrency analysis proceeds in $O(R^3)$ time where R is the number of regions in the program [8].

5. Maintaining Precision

The precision of the provided information is a critical issue, as the analysis to be of practical use must not be overly conservative. There are some sources in the analysis that may introduce imprecision, and we first describe two cases where we can easily increase the precision. We then discuss other sources and give suggestions to resolve the potentially introduced imprecision.

The purpose of the activation context analysis is to determine regions that execute before and after each activation of a unit U in the sets $before(U.start)$ and $after(U.stop)$ and then distribute these sets throughout unit U. Multiple activation sites are handled conservatively by taking the intersection over the *before* and *after* sets of individual sites. However, there are two cases where a deeper investigation of the activation sites can lead to more precise information. These cases are multiple ordered sites and recursion.

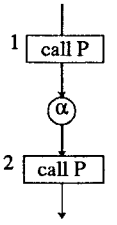


Fig. 7: Ordered activation sites

For multiple ordered activation sites, the regions executing between any of the activation sites are ignored in the computed ordering sets. Consider Fig. 7 showing a procedure P with only two activation sites in regions 1 and 2 such that $1 \in before(2)$. Then, the regions that execute *between* the two calls, i.e., the regions in $after(1) \cap before(2)$, which in this case is region α, will neither occur in $before(P.start)$ nor in

after (*P.stop*). But for each of the two instances of P, α executes either before or after P, and thus α is ordered with all regions inside P. We can generalize the above observation for any set of ordered activation sites to a unit U and determine the regions that execute *between* any of the activations of U. We improve the computed *Ord* sets by adding these regions to the initial estimate Ord_{init}.

Recursion is another case where refinements are possible. Consider the recursive procedure P in Fig. 8 (i). Assuming the two call sites shown in the figure are the only call sites to P, the intersection of their *before* sets will be empty, i.e., *before* (*P.start*) = ∅. Thus, the computation of *before* (*P.start*) does not recognize the fact that region r actually executes before any instance of P. This additional ordering can be detected by distinguishing recursive call sites. In Fig. 8, the recursive call site in region 2 can never initiate the recursion. Thus, a region that is not included in the *before* set of region 2 (e.g., region r in the figure) is still guaranteed to execute before any instance of procedure P, if it is contained in the *before* sets of all activation sites that can possibly initiate the recursion. It follows, that *before* (*P.start*) can simply be determined as *before* (1) ∪ {1} in Fig. 8.

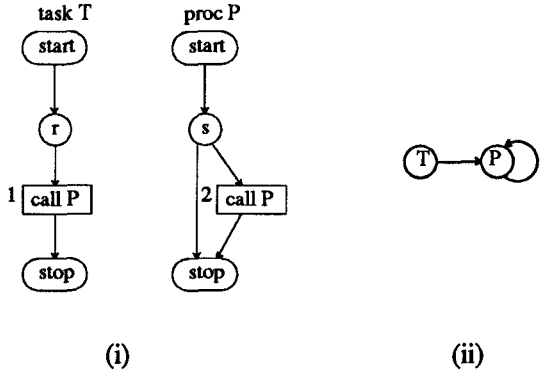


Fig. 8: A recursive procedure and the call graph

In general, we need to identify the recursive call sites that can never initiate the recursion. We refer to such a non-initiating recursive call site as a *must-recursive* site. Once the must-recursive activation sites are determined, we improve the computation of *before* (*U.start*) for a unit U by excluding the must-recursive sites from the intersection. Must-recursive call sites are detected by analyzing the program's call graph for dominators [1]. Intuitively, an edge $e=(p,q)$ in the call graph representing a call to unit q from unit p is a must-recursive edge if node p cannot be reached without traversing through node q. Thus, an edge $e=(p,q)$ is a must-recursive edge if and only if its sink q dominates its source p. Consider the call graph in Fig. 7 (ii). As every node dominates itself, the edge (P, P) is a must-recursive edge. Detecting dominators in a graph can be done in almost linear time [9]. Below, we show the refined equation for determining the activation context of a unit U in the set *before* (*U.start*).

$$before(U.start) = \bigcap_{\substack{s \text{ an activation site} \\ s \text{ not must-recursive}}} [\, before(s) \cup gen(s)]$$

Another source of imprecision is the presence of spurious synchronization edges. A synchronization edge is spurious if the execution order it enforces is already enforced by other ordering mechanisms. For example, if such a spurious edge occurs in the computation of an intersection for an ordering set, the presence of the edges may actually reduce the precision of the computed information. A naive approach to identify and remove spurious synchronization edges simply removes an edge and then determines whether the resulting ordering does at least include the ordering in the presence of the edge, in which case the edge was spurious. We are currently investigating more efficient ways to identify and remove spurious synchronization edges.

6. Applications

The computed execution order has applications in various program development tools. We first show how the computed information is used for analyzing a parallel program for data races.

6.1. Data Race Detection

Data races occur if two tasks access a shared variable in an unpredictable order. There are two classes of data races: write/write (w/w)-races, where two tasks simultaneously update a shared variable and read/write (r/w)-races, where a shared variable is used and defined by two tasks simultaneously. Once the partial region order is available, data races can easily be determined by examining the definitions and uses of shared variables among unordered regions.

Claim: Let $Def(r)$ be the set of shared variables defined in region r, and $Use(r)$ the set of shared variables that are used in r. A data race involving the shared variable x may exist between two regions r and s, if and only if

(1) $r \notin Ord(s)$ or $s \notin Ord(r)$ and

(2) $x \in (Def(r) \cap Def(s))$ (w/w-race) or

$x \in (Def(r) \cap Use(s)) \cup (Use(r) \cap Def(s))$ (r/w-race)

Proof: Follows immediately from the definition of a data race and the correctness of the *Ord* sets.

The determination of the sets $Def(r)$ and $Use(r)$ may be affected by aliasing in the program. Thus if there is a definition of a variable x in region r, then not only should x be added to $Def(r)$, but also all its aliases should be added. Existing alias detection techniques [6] can easily be incorporated to provide the necessary alias information.

region	*Def*	*Use*	concurrent defs & uses
task Main			
2	y	-	-
6	-	y	11,14
task T			
9	y	-	-
11	-	y	6,14
proc P			
14	y	y	6,11,14

Table 5

We illustrate the data race detection for our example from Fig. 2. The regions of interest in this example, i.e., regions that contain definitions and uses of variables, are regions 2,6,9,11 and 14. There is only one shared variable (variable y) in this example. Interprocedural alias detection techniques [6] can determine that the formal parameter v of procedure P is an alias for the shared variable y. Thus we obtain the sets *Def* and *Use* as shown in Table 5. The last column in Table 5 is extracted from Table 4 and shows the concurrent regions that contain definitions or uses of the shared variable y.

From Table 5 we can determine a potential r/w-race for the two pairs of regions (6,14) and (11,14). As region 14 may execute concurrently with itself, we also determine potential w/w-races and r/w-races involving variable y for the use and the two definitions in region 14.

6.2. Other Applications

Although we have not specifically addressed the problem of detecting synchronization anomalies or infinite-waits (the two terms are used here interchangeably), the computed concurrency information can naturally be used to detect a subclass of infinite-waits. Intuitively, an infinite-wait occurs when a set of tasks is waiting for a rendezvous to take place, either at an accept or at an entry call statement, but the rendezvous cannot possibly occur. An infinite-wait is called a *deadlock* if the set of waiting tasks forms a cyclic chain. The class of infinite-wait anomalies that can be detected using only the computed concurrency information are deadlocks that are guaranteed to occur in every program execution. We refer to this class of deadlocks as *static deadlocks*. A static deadlock is reflected in the MIG as a cycle consisting of alternating synchronization edges and control flow paths. The presence of such a cycle causes a circular region propagation during the synchronization analysis, leading to regions being propagated to their own *before* sets. Thus, during the synchronization analysis, whenever a region r enters its own *before* set. i.e., $r \in before(r)$, we report a static deadlock.

In general, the detection of infinite-wait anomalies, including deadlocks as a special case, has been shown to be NP-complete [16]. Recently, a polynomial time approximation for deadlock detection [12] has been developed that incorporates a relation similar to our *before* relation into the detection analysis.

The applications of static concurrency analysis are however not limited to debugging. The computed region ordering can also be used to evaluate the degree of parallelism in a concurrent program by relating the amount of ordered events to the unordered ones. In a similar context, information about concurrency can be used to guide parallelizing transformations. Some transformations in parallelizing compilers, such as loop unrolling, are typically applied based on heuristics. Evaluating the parallelism as it results from a parallelizing transformation may be used to guide the application of such transformations. Finally, the execution order can be used to enable conventional program optimizations in a concurrent program as it computes the order of uses and

definitions of shared variables across task boundaries.

7. Conclusion and Related Work

We have presented a data-flow based concurrency analysis technique for the automatic detection of data races in parallel programs. By formulating the problem in a data-flow framework, the technique computes a partial execution order of statements. Unlike previous methods in data-flow based concurrency analysis, the technique efficiently incorporates procedures into the analysis without placing restrictions on the use of procedures.

Data flow analysis techniques are very efficient and operate in low order polynomial time. Employing data-flow analysis techniques to compute an execution order of statements has been used by several authors. [3-5, 14] However, these techniques have not addressed interprocedural effects on the concurrency analysis. Certainly, procedures can be incorporated by in-line code substitution. However, the size of the graphs increases tremendously and recursion would not be allowed. Among the data-flow techniques, the one of most relevance to our work [4, 5] is applied to a parallel nonprocedural language with low-level synchronization that may be used to synchronize parallel loops and parallel case statements.

A different approach to static concurrency analysis is taken in state based techniques. Taylor [15] presents a state based analysis that generates possible concurrency states in a program, whereas a concurrency state represents the communication status of all tasks. By the nature of simulation, state based analysis is an inherently costly analysis and several suggestions to reduce the number of generated states have been made. Young and Taylor [17] combine state based analysis with symbolic execution to rule out unreachable concurrency states. McDowell and Appelbe [2, 13] propose to summarize similar concurrency states into *clans*. Reducing the number of generated states by using a different graphical representation, the *task interaction graphs*, was proposed by Long and Clarke. [10, 11]

The basic difference between the data-flow based and state based techniques for concurrency analysis is the trade off between precision and analysis cost. State based techniques are likely to generate more precise information than data-flow analysis techniques but the concurrency analysis is more costly to perform, being exponential in the worst case. Data flow based analysis is efficient in that it has a low order polynomial time complexity. Interprocedural techniques for computing global data-flow information are being used in programming tools, indicating that, at least for these uses, the imprecision in these techniques does not cause major problems.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman, in *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Massachusetts, 1986.
2. W. F. Appelbe and C. E. McDowell, "Anomaly reporting: A tool for debugging and developing parallel numerical algorithms," *Proceedings of the First International Conference on Supercomputing Systems, IEEE*, pp. 386-391, 1985.
3. G. Bristow, C. Drey, B. Edwards, and W. Riddle, "Anomaly detection in concurrent programs," *Proceedings of Fourth International Conference on Software Enginering, IEEE*, pp. 265-273, Munich, Germany, 1979.
4. D. Callahan and J. Subhlok, "Static analysis of low-level synchronization," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, vol. 24, no. 1, pp. 100-111, January 1989.
5. D. Callahan, K. Kennedy, and J. Subhlok, "Analysis of event synchronization in a parallel programming tool," *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 21-30, Seattle WA, March 1990.
6. K. D. Cooper, "Analyzing aliases of reference formal parameters," *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pp. 281-290, New Orleans, Louisiana , January 1985.
7. United States Department of Defense, "Reference Manual for the Ada Programming Language ," *(ANSI/MIL-STD-1815A)*, Washington, D.C., January 1983.
8. E. Duesterwald, "Static concurrency analysis in the presence of procedures," *Technical Report #91-6*, Dept. of Computer Science, University of Pittsburgh, March 1991.
9. T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph,"

ACM Transactions on Programming Languages and Systems, vol. 1, no. 1, pp. 121-141, July 1979.

10. D. Long and L. Clarke, "Task interaction graphs: a representation for concurrency analysis," *Technical Report*, Computer Science Department, University of Massachusetts, Amherst, March 1988.
11. D. Long and L. Clarke, "Task interaction graphs for concurrency analysis," *Proceedings of the Eleventh International Conference on Software Enginering*, pp. 44-52, Pittsburgh, Pennsylvania , May 1989.
12. S. Masticola and B. G. Ryder, "A model of Ada programs for static deadlock detection in polynomial time," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 91-102, Santa Cruez, California, May 1991.
13. C. McDowell, "A practical algorithm for static analysis of parallel programs," *Journal of Parallel and Distributed Computing*, vol. 6, no. 3, pp. 515-536, 1989.
14. R. N. Taylor and L. J. Osterweil, "Anomaly detection in concurrent software by static data flow analysis," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 265-278, May 1980.
15. R. N. Taylor, "A general purpose algorithm for analyzing concurrent programs," *Communications of the ACM*, vol. 26, no. 5, pp. 362-376, May 1983.
16. R. N. Taylor, "Complexity of analyzing the synchronization structure of concurrent programs," *Acta Informatica*, vol. 19, no. 1, pp. 57-83, 1983.
17. M. Young and R. N. Taylor, "Combining static concurrency analysis with symbolic execution," *Proceedings of Workshop on Software Testing*, pp. 10-18, 1986.